ORBITER
Programming Interface

©2001-2002 Martin Schweiger
martins@medphys.ucl.ac.uk
www.medphys.ucl.ac.uk/~martins/orbit/orbit.html
19 April 2002

1. Introduction

This reference document contains the specification for the Orbiter Programming Interface. It is not required for running Orbiter.
The programming interface allows the development of third party modules to enhance the functionality of the Orbiter core. Examples for modules are:

- Additional instruments, simulation monitoring devices, and spacecraft controls
- Custom flight models
- Custom instrument panels (planned)
- Multiplayer modules
- Custom calculation of planetary positions

The API is in an early stage of development. Future versions will probably change significantly, mainly by expanding the list of supported functions.

2. Requirements

The following components are required to build an addon module:

- The latest Orbiter package
- The Orbiter SDK libraries and include files (contained in the Orbiter SDK package)
- A C++ compiler running under Windows (the SDK was developed with VC++, but other compilers should also work)

3. Preparation

- Install the Orbiter package, if you haven't already done so.
- Install the Orbiter SDK package. This will generate the *OrbiterSDK* subdirectory containing the header files and libraries required for building plugins.
- Create a project for your plugin DLL (the method depends on the compiler used). Make sure you use thread-safe system libraries ("Multithread DLL"). Add *OrbiterSDK\include* to the include search path, and add *OrbiterSDK\lib\Orbiter.lib* and *OrbiterSDK\lib\Orbitersdk.lib* to the link stage.
- Write the code for your plugin, compile and link it, and move the resulting DLL to the *Orbiter\Modules\Plugin* folder.
- Run Orbiter, go to the *Modules* tab in the launchpad dialog, and activate your new plugin.

4. SDK files

The following files are contained in the Orbiter development kit:

Orbitersdk\doc*	*SDK documentation*
Orbitersdk\include	
Orbitersdk.h	*The interface header file*
Orbitersdk\lib	
Orbitersdk.lib	*The DLL auxiliary library*
Orbiter.lib	*The Orbiter API library*
Orbitersdk\tools*	*Tools for model and texture generation*
Orbitersdk\samples*	*Sample source code*

5. Concept

Definition of terms used in this document:

Module

A *module* is a dynamic link library (DLL) which extends or replaces functionality of the core Orbiter program. Modules interact with Orbiter via callback functions conforming to the public interface defined below.

Plugin

Plugins are generic modules not linked to any particular object. They may include popup windows for displaying or manipulating general simulation information, multiplayer interfaces, etc. Plugins can be activated or deactivated by the user via the Modules tab in the Orbiter Launchpad dialog.

Planet module

Planet modules are linked to planets or moons and are used specifically for updating planetary position and velocity data. Planet modules are referenced via the planet/moon's configuration file.

Vessel module

Vessel modules are linked to specific spacecraft, to allow customisation of the vessel's behaviour. Vessel modules are referenced via the vessel class configuration file.

In all active modules, Orbiter executes *callback functions* corresponding to certain simulation conditions. For example, whenever the simulation window is opened after the user presses the *Orbiter* button in the launchpad dialog, Orbiter calls the *opcOpenRenderViewport* callback function in all plugins to allow initialisation routines to be performed. A plugin doesn't need to implement all callback functions defined in the interface. However, the programmer is responsible for implementing callback functions in a consistent way. For example, if the plugin allocates memory for data in *opcOpenRenderViewport*, then this memory should be deallocated in *opcCloseRenderViewport*. The SDK allows access to core parts of the Orbiter simulator, and bugs in active plugins may cause the program to crash.

All callback functions use a C stack frame, so they need to be defined as *extern "C"* for compilation with a C++ compiler. For convenience the *DLLCLBK* macro is provided in Orbitersdk.h to use as modifier for callback function definitions.

The code for the callback functions may contain calls to the Orbiter API functions, to obtain and set simulation parameters such as object positions and speed, simulation time, etc. API functions use an oapi ("orbiter API") prefix. API functions use a C++ stack frame.

6. Some useful hints

- Your plugin should not open popup windows or dialogs outside the render window when running in fullscreen mode. Check the fullscreen flag passed to the `opcOpenRenderViewport` callback function to see whether it is safe to open a window.
- If you intercept a callback function which is called at each frame (like `opcTimestep`), make it as efficient as possible, or simulation performance will suffer.

7. Sample modules

The Orbitersdk\samples folder contains a few projects which can be used as a starting point for creating your own plugins. To compile a sample using VC++:

- Load the project file (*.dsw) into VC++.
- Build the project.
- Copy the DLL from the Debug or Release subdirectory into the Orbiter\Modules\Plugin directory (plugins) or into the Orbiter\Modules directory (planet and vessel modules).
- To activate new plugins, run Orbiter, activate the plugin under the Modules tab, and launch the simulation.

- New planet or vessel modules are used automatically if they are referenced by the relevant definition files.

Warpcontrol
Opens a dialog which allows fine-tuning of time acceleration. (available only in window mode)

Rcontrol
Opens a dialog which allows to switch and control spacecraft on the fly. (available only in window mode)

Atlantis
The complete code for Orbiter's reference implementation of the Atlantis (Space Shuttle) module, including modules for post-separation SRBs (solid rocket boosters) and main tank.

8. Programming techniques

8.1. Defining an animation sequence for a vessel

Animation sequences can be used to simulate movable parts of a vessel. Examples are the deployment of landing gear, cargo door operation, or animation of airfoils.
Animations are implemented in *vessel modules*, using the VESSEL interface class.
Orbiter allows two types of animation: rotation and translation. More complex can be built from these basic operations.

Mesh requirements:
Animations are performed by transforming mesh groups. Therefore, all parts of the mesh participating in an animation must be defined in separate groups. If multiple groups participate in a single transformation, those groups must be stored in sequence in the mesh.

Module prerequisites:
If it doesn't exist already, create a C++ project for the vessel module.
Derive a class from VESSEL, e.g.

```
class MyVessel: public VESSEL {
  // ...
};
```

Implement the `ovcInit` and `ovcExit` callback functions to create and destroy an instance of `MyVessel`, e.g.

```
DLLCLBK VESSEL *ovcInit (OBJHANDLE hvessel, int flightmodel)
{
  return new MyVessel (hvessel, flightmodel);
}

DLLCLBK void ovcExit (VESSEL *vessel)
{
  delete (MyVessel*)vessel;
}
```

Defining an animation sequence:
Create a member function for MyVessel to define animation sequences, and call it from the constructor, e.g.

```
MyVessel::MyVessel (OBJHANDLE hObj, int fmodel)
: VESSEL(hObj, fmodel)
{
  DefineAnimations();
}
```

In the body of DefineAnimations(), you now have to specify how the animation should be performed. Here is an example for a nose wheel animation:

```
void MyVessel::DefineAnimations()
```

```
{
  static UINT groups[4] = {5,6,10,11}; // participating groups
  static ANIMCOMP nosewheel = {
    groups, 4,                  // group list and # groups
    0.3, 1.0,                   // limiting states
    0,-1.0,8.5,                 // rotation reference
    1.0,0.0,0.0,                // rotation axis
    (float)(-0.5*PI),           // rotation range
    0,                          // mesh no.
    0,                          // not used
    MESHGROUP_TRANSFORM::ROTATE // transform type
  };

  anim_gear = RegisterAnimSequence (0.0);
  AddAnimComp (anim_gear, &nosewheel);
}
```

The ANIMCOMP structure defines all the parameters of the animation. In this case:

groups	pointer to a list of indices (starting with 0) of mesh groups participating in the animation.
4	defines the number of groups to be part of the animation (must be ≥ 1).
0.3, 1.0	specifies that the animation should take place between states 0.3 and 1.0 of the animation sequence. This means that the wheel will remain fully retracted between states 0.0 and 0.3, and will gradually be deployed between states 0.3 and 1.0. This allows us to perform a different animation first, for example open the gear doors.
0,-1.0,8.5	defines a reference point (in the vessel's frame of reference) for the rotation.
1.0,0.0,0.0	defines the axis around which the rotation takes place.
(float)(-0.5*PI)	the angular range over which the rotation is performed (in radians)
0	the mesh index (starting with 0 for the vessel's first mesh)
0	the ngroup member of the MESHGROUP_TRANSFORM struct which is not used here.
MESHGROUP_TRANSFORM::ROTATE	the animation type (rotation)

RegisterAnimSequence() defines a new sequence. The sequence identifier is stored in anim_gear. Parameter "0.0" indicates that the wheel is defined in its retracted state in the mesh.
AddAnimComp() adds the nosewheel animation to the sequence.

Additional animations can be added to the same sequence by defining additional ANIMCOMP structures and adding them with AddAnimComp().

Additional sequences (for example to animate cargo doors) can be added by additional RegisterAnimSequence() calls.

Note that all ANIMCOMP structures must be defined static because Orbiter does not create a local copy. This means that the animations are global properties of the vessel class.

Performing the animation:
To animate the nose wheel now, we need to manipulate the animation sequence state by calling SetAnimState() with a value between 0 (fully retracted) and 1 (fully deployed). This is typically done in the Timestep() member function, e.g.

```
void MyVessel::Timestep (double simt)
{
  if (gear_status == CLOSING || gear_status == OPENING) {
    double da = oapiGetSimStep() * gear_speed;
    if (gear_status == CLOSING) {
      if (gear_proc > 0.0)
        gear_proc = max (0.0, gear_proc-da);
      else
```

```
        gear_status = CLOSED;
    } else  { // door opening
      if (gear_proc < 1.0)
        gear_proc = min (1.0, gear_proc+da);
      else
        gear_status = OPEN;
    }
    SetAnimState (anim_gear, gear_proc);
  }
}
```

Here, gear_status is a flag defining the current operation mode (CLOSING, OPENING, CLOSED, OPEN). This will typically be set by user interaction, e.g. by pressing a keyboard button. If the animation is in progress (OPENING or CLOSING), we determine the rotation step (da) as a function of the current frame interval (oapiGetTimeStep()). The value of gear_speed defines how fast the gear is deployed.
Next, we update the deployment state (gear_proc), and check whether the sequence is complete (≤0 if closing, or ≥1 if opening). Finally, SetAnimState() is called to perform the animation.

The DeltaGlider sample module (Orbitersdk\samples\DeltaGlider) contains a complete example for an animation implementation.

8.2. Manual animation

As an alternative to the (semi-)automatic animation concept described in the previous section, Orbiter also allows manual animation. This can be more versatile, but requires more effort from the module developer, because the complete animation sequence must be implemented explicitly.
A manual animation sequence is created by the functions `VESSEL::RegisterAnimation()` and `VESSEL::UnregisterAnimation()`. A call to `RegisterAnimation` causes Orbiter to call the module's `ovcAnimate` callback function at each frame, provided the vessel's visual exists. `UnregisterAnimation` cancels the request.
Note that `RegisterAnimation`/`UnregisterAnimation` pairs can be nested. Each call to `RegisterAnimation` increments a reference counter, each call to `UnregisterAnimation` decrements the counter. Orbiter will call `ovcAnimate` as long as the counter is > 0.
It is up to the module to implement its animations in the body of `ovcAnimate`. Typically this will involve calls to `MeshgroupTransform()`, to rotate or translate mesh groups as a function of the last simulation time step. Note that `ovcAnimate` is called only once per frame, even if more than one `RegisterAnimation` request has been logged. The module must therefore decide which animations need to be processed in `ovcAnimate`.
`UnregisterAnimation` should never be called from inside `ovcAnimate`, since `ovcAnimate` is only called if the visual exists. This could cause the unregister request to be lost. It is better to test for animation termination in `ovcTimestep`.

9. Data types

OBJHANDLE

A handle for a logical object. Objects can be vessels, orbital stations, spaceports, planets, moons or suns.

VISHANDLE

A handle for a visual object. These are representations for logical objects for the purpose of rendering. Visuals exist only if the object is within visual range of the camera, and are created and deleted as needed.

MESHHANDLE

A handle for object meshes.

SURFHANDLE

A handle for a bitmap surface. Surfaces are currently used for drawing instrument panel areas.

VECTOR3

Double precision vector $\in R^3$

Synopsis:

```
typedef union {
   double data[3];
   struct { double x, y, z; };
} VECTOR3;
```

MATRIX3

Double precision matrix $\in R^{3x3}$

Synopsis:

```
typedef union {
  double data[9];
  struct { double m11, m12, m13,
                  m21, m22, m23,
                  m31, m32, m33; };
} MATRIX3;
```

ELEMENTS

Keplerian orbital elements

Synopsis:

ENGINESTATUS

Defines the thruster status for a spacecraft

Synopsis:

```
struct {
  double main;      main/retro thruster level [-1,+1]
  double hover;     hover thruster level [0,+1]
  int attmode;      attitude thruster mode [0=rot, 1=lin]
} ENGINESTATUS;
```

ENGINETYPE

Enumerates thruster types

Synopsis:

```
typedef enum {
  ENGINE_MAIN,
  ENGINE_RETRO,
  ENGINE_HOVER,
  ENGINE_ATTITUDE
} ENGINETYPE;
```

EXHAUSTTYPE

Enumerates engine groups for exhaust rendering.

Synopsis:

```
typedef enum {
  EXHAUST_MAIN,
  EXHAUST_RETRO,
  EXHAUST_HOVER,
  EXHAUST_CUSTOM
} EXHAUSTTYPE;
```

VESSELSTATUS

Defines vessel status parameters at a given time.

Synopsis:

```
typedef struct {
   VECTOR3 rpos;
   VECTOR3 rvel;
   VECTOR3 vrot;
   VECTOR3 arot;
   double fuel;
   double eng_main;
   double eng_hovr;
   OBJHANDLE rbody;
   OBJHANDLE base;
   int port;
   int status;
   VECTOR3 vdata[10];
   double fdata[10];
   DWORD flag[10]
} VESSELSTATUS;
```

rpos	position relative to reference body in ecliptic frame (see notes)
rvel	velocity relative to reference body in ecliptic frame
vrot	rotation velocity about principal axes in ecliptic frame
arot	vessel orientation against ecliptic frame
fuel	fuel level [0...1]
eng_main	main engine setting [-1...1]
eng_hovr	hover engine setting [0...1]
rbody	handle of reference body
base	handle of docking or landing target
port	designated docking or landing port
status	0=freeflight, 1=landed, 2=taxiing, 3=docked, 99=undefined
vdata	vector buffer for future extensions. Currently used: vdata[0] contains landing parameters if status==1: vdata[0].x = longitude [rad], vdata[0].y = latitude [rad] of landing site, vdata[0].z = orientation of vessel [rad].
fdata	scalar buffer for future extensions. Not currently used.
flag	integer buffer for future extensions. Not currently used.

Notes:

- arot=(α,β,γ) contains angles of rotation [rad] around x,y,z axes in ecliptic frame to produce this rotation matrix **R** for mapping from the vessel's local frame of reference to the global frame of reference:

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  such that

$$\mathbf{r}_{global} = \mathbf{R}\,\mathbf{r}_{local} + \mathbf{p}$$

  where **p** is the vessel's global position.

10. Constants

Navmode constants

NAVMODE_KILLROT	*engage attitude thrusters to kill rotation*
NAVMODE_HLEVEL	*engage attitude thrusters to keep level with horizon*
NAVMODE_PROGRADE	*engage attitude thrusters to turn prograde*
NAVMODE_RETROGRADE	*engage attitude thrusters to turn retrograde*
NAVMODE_NORMAL	*engage attitude thrusters to turn orbit-normal*

NAVMODE_ANTINORMAL	*engage attitude thrusters to turn orbit-antinormal*
NAVMODE_HOLDALT	*engage hover thrusters to maintain altitude*

HUD mode constants

HUD_NONE
HUD_ORBIT
HUD_SURFACE
HUD_DOCKING

MFD mode constants

MFD_NONE
MFD_ORBIT
MFD_SURFACE
MFD_MAP
MFD_LANDING
MFD_DOCKING
MFD_OPLANEALIGN
MFD_OSYNC
MFD_TRANSFER
MFD_USERTYPE

MFD identifier constants

MFD_LEFT
MFD_RIGHT
MFD_USER1
MFD_USER2
MFD_USER3

11. Class VESSEL

This class constitutes the interface with Orbiter's internal vessel implementation, and provides access to the various status parameters and methods of individual spacecraft. Typically, an instance of VESSEL or a derived class will be constructed in each vessel module. An example for a typical application of the VESSEL class can be found in the Atlantis sample code.

Public member functions

11.1. Construction/creation

VESSEL

Constructor. Creates a vessel from a vessel handle.

Synopsis:

```
VESSEL (OBJHANDLE hVessel, int flightmodel)
```

Parameters:

hVessel	vessel handle
flightmodel	level of realism requested. (0=simple, 1=realistic)

Create

This function is obsolete and has been replaced by oapiCreateVessel.

GetHandle

Returns the Vessel handle.

Synopsis:

```
const OBJHANDLE GetHandle (void) const
```

Return value:

vessel handle, as passed to the constructor.

Notes:

- The handle is useful for various API function calls.

11.2. Vessel parameters and capabilities

GetName

Returns the vessel's name.

Synopsis:

```
char *GetName (void) const
```

Return value:

Pointer to vessel's name.

GetClassName

Returns the vessel's class name.

Synopsis:

```
char *GetClassName (void) const
```

Return value:

Pointer to vessel's class name.

GetFlightModel

Returns the requested realism level for the flight model.

Synopsis:

```
int GetFlightModel (void) const
```

Return value:

Realism level. These values are currently supported:
0 = simple
1 = realistic

GetSize

Returns the vessel's mean radius.

Synopsis:

```
double GetSize (void) const
```

Return value:

Vessel mean radius [m].

GetEmptyMass

Returns vessel's empty mass excluding fuel and payload. Equivalent to the oapiGetEmptyMass API function.

Synopsis:

```
double GetEmptyMass (void) const
```

Return value:

Vessel empty mass [kg].

GetMaxFuelMass

Returns vessel's maximum fuel capacity. Equivalent to the oapiGetMaxFuelMass API function.

Synopsis:

```
double GetMaxFuelMass (void) const
```

Return value:

Vessel maximum fuel mass [kg].

GetISP

Returns vessel's fuel-specific impulse.

Synopsis:

```
double GetISP (void) const
```

Return value:

Fuel-specific impulse [m/s]. This is the amount of thrust [N] obtained by burning 1kg of fuel per second.

GetMaxThrust

Returns maximum thrust rating [N] for one of the vessel's engine groups, defined by *eng*.

Synopsis:

```
double GetMaxThrust (ENGINETYPE eng) const
```

Parameters:

eng engine group identifier

Return value:

Maximum thrust rating [N]

GetMainThrustModPtr

Returns the address of a variable which can be used to define custom main/retro thrust [Newton] outside the control of the Orbiter core. Orbiter includes this value in the acceleration calculation, but the module is responsible for adjusting the corresponding physical parameters, like mass changes due to fuel consumption. Negative values define retro thrust.

Synopsis:

```
double *GetMainThrustModPtr (void) const
```

Return value:

Pointer to a variable which the module can set to alter orbiter's main/retro thrust setting. The value of this variable is added to the result of the standard thrust calculation.

GetCOG_elev

Returns the altitude of the vessel's centre of gravity over ground level when landed [m].

Synopsis:

```
double GetCOG_elev (void) const
```

Return value:

elevation of vessel's centre of mass [m].

GetCrossSections

Returns the vessel's cross sections projected in the direction of the vessel's principal axes [m^2]

Synopsis:

```
void GetCrossSections (VECTOR3 &cs) const
```

Parameters:

cs — vector receiving the cross sections of the vessel's projection into the y-z, x-z, and x-y planes, respectively [m^2]

GetCW

Returns the vessel's wind resistance coefficients in the principal directions [dimensionless].

Synopsis:

```
void GetCW (
     double &cw_z_pos,
     double &cw_z_neg,
     double &cw_x,
     double &cw_y) const
```

Parameters:

cw_z_pos	resistance in positive z direction (forward)
cw_z_neg	resistance in negative z direction (back)
cw_x	resistance in lateral direction
cw_y	resistance in vertical direction

Notes:

- The first value (cw_z_pos) is the coefficient used if the vessel's airspeed z-component is positive (vessel moving forward). The second value is used if the z-component is negative (vessel moving backward).
- Lateral and vertical components are assumed symmetric.

GetWingAspect

Returns the vessel's wing aspect ratio (wingspan^2 / wing area). Vessels without wing-type airfoils return 0.

Synopsis:

```
double GetWingAspect (void) const
```

Return value:

Wing aspect ratio (wingspan^2 / wing area)

GetWingEffectiveness

Returns wing form factor: ~3.1 for elliptic wings, ~2.8 for tapered wings, ~2.5 for rectangular wings.

Synopsis:

```
double GetWingEffectiveness (void) const
```

Return value:

Wing form factor.

Notes:

- This form factor describes the wing's effectiveness in producing lift in an atmosphere as a function of its shape.

GetRotDrag

Returns the vessel's resistance $r_{x,y,z}$ against rotation around axes in atmosphere.

Synopsis:

```
void GetRotDrag (VECTOR3 &rd) const
```

Parameters:

rd — rotational drag coefficient in the three coordinate axes of the vessel's frame of reference.

Notes:

- rd contains the components $r_{x,y,z}$ against rotation around axes in atmosphere, where angular deceleration due to atmospheric friction is $a^{(\omega)}_{x,y,z} = -v^{(\omega)}_{x,y,z}\ \rho\ r_{x,y,z}$ with angular velocity $v^{(\omega)}$ and atmospheric density ρ.

GetPMI

Returns principal moments of inertia, mass-normalised [m^2]

Synopsis:

```
void GetPMI (VECTOR3 &pmi) const
```

Parameters:

pmi	Diagonal elements of the inertia tensor

Notes:

For the meaning of the pmi vector, see `SetPMI`.

GetCameraOffset

Returns the camera position for internal (cockpit) view.

Synopsis:

```
void GetCameraOffset (VECTOR3 &ofs) const
```

Parameters:

ofs	camera offset in the vessel's local frame of reference [m,m,m]

SetSize

Sets the vessel's mean radius [m].

Synopsis:

```
void SetSize (double size) const
```

Parameters:

size	vessel mean radius [m]

Notes:

- This value is used for visibility calculations, but normally has no influence on the actual visual representation of the object (which is defined by the mesh) unless the module performs mesh scaling operations.

SetEmptyMass

Sets the vessel's empty mass excluding fuel and payload. Equivalent to the oapiSetEmptyMass API function.

Synopsis:

```
void SetEmptyMass (double m) const
```

Parameters:

m	vessel empty mass [kg]

SetMaxFuelMass

Sets the vessel's maximum fuel capacity.

Synopsis:

```
void SetMaxFuelMass (double m) const
```

Parameters:

m	Maximum fuel mass [kg].

SetISP

Sets the vessel's fuel-specific impulse.

Synopsis:

```
void SetISP (double isp) const
```

Parameters:

isp	fuel-specific impulse [m/s].

Notes:

- The ISP defines the amount of thrust [N] obtained by burning 1 kg of fuel per second.

SetMaxThrust

Sets the maximum thrust rating for engine group *eng* to *th* [N].

Synopsis:

```
void SetMaxThrust (ENGINETYPE eng, double th) const
```

Parameters:

eng	engine group identifier
th	maximum thrust rating [N]

SetCOG_elev

Sets the altitude of the vessel's centre of gravity over ground level when landed [m].

Synopsis:

```
void SetCOG_elev (double h) const
```

Parameters:

h	elevation of the vessel's centre of gravity above the surface plane when landed [m].

Notes:

- This function is obsolete and has been replaced by SetTouchdownPoints.

SetTouchdownPoints

This defines 3 surface contact points for ground contact calculations (e.g. the points where the landing gear touches the ground).

Synopsis:

```
void SetTouchdownPoints (
     const VECTOR3 &pt1,
     const VECTOR3 &pt2,
     const VECTOR3 &pt4) const
```

Parameters:

pt1	touchdown point of nose wheel (or equivalent)
pt2	touchdown point of left wheel (or equivalent)
pt3	touchdown point of right wheel (or equivalent)

Notes:

- The points are the positions at which the vessel's undercarriage (or equivalent) touches the surface, specified in local vessel coordinates.
- The points should be specified such that the cross product pt3-pt1 x pt2-pt1 defines the horizon UP direction for the landed vessel (given a left-handed coordinate system).

SetCrossSections

Sets the vessel's cross sections projected in the direction of the vessel's principal axes [m^2].

Synopsis:

```
void SetCrossSections (const VECTOR3 &cs) const
```

Parameters:

cs	vector of cross sections of the vessel's projection into the y-z, x-z, and x-y planes, respectively [m^2]

SetCW

Sets the vessel's wind resistance coefficients along the local reference axes [dimensionless].

Synopsis:

```
void SetCW (
     double cw_z_pos,
     double cw_z_neg,
     double cw_x,
     double cw_y) const
```

Parameters:

cw_z_pos	resistance in positive z direction (forward)
cw_z_neg	resistance in negative z direction (back)
cw_x	resistance in lateral direction
cw_y	resistance in vertical direction

Notes:

- The first value (cw_z_pos) is the coefficient used if the vessel's airspeed z-component is positive (vessel moving forward). The second value is used if the z-component is negative (vessel moving backward).
- Lateral and vertical components are assumed symmetric.

SetWingAspect

Sets the wing aspect ratio (wingspan^2 / wing area).

Synopsis:

```
void SetWingAspect (double aspect) const
```

Parameters:

aspect	wing aspect ratio [dimensionless]

Notes:

- This value is used for atmospheric drag calculation.
- Only vessels with wing-type airfoils should call this function.

SetWingEffectiveness

Sets the wing form factor. Used for lift and drag calculation.

Synopsis:

```
void SetWingEffectiveness (double we) const
```

Parameters:

we	wing form factor.

Notes:

- Typical values are: ~3.1 for elliptic wings, ~2.8 for tapered wings, ~2.5 for rectangular wings.

void SetRotDrag (const VECTOR3&) const

Sets the vessel's resistance against rotation around axes in atmosphere.

void SetPitchMomentScale (double scale) const

Sets the magnitude of the moment acting on the vessel's pitch angle which rotates the vessel's longitudinal direction towards the airspeed vector.

void SetBankMomentScale (double scale) const

Sets the magnitude of the moment acting on the vessel's bank angle which rotates the vessel's longitudinal direction towards the airspeed vector.

SetPMI

Sets principal moments of inertia, mass-normalised [m^2].

Synopsis:

```
void SetPMI (const VECTOR3 &pmi) const
```

Parameters:

pmi Principal moments of inertia

Notes:

- The principal moments are the diagonal elements of the inertia tensor in a frame of reference where the off-diagonal elements are zero.
- The elements of pmi should be calculated as follows:

$$pmi_1 = \frac{1}{M}\int \rho(r)(r_y^2 + r_z^2)dr$$

$$pmi_2 = \frac{1}{M}\int \rho(r)(r_x^2 + r_z^2)dr$$

$$pmi_3 = \frac{1}{M}\int \rho(r)(r_x^2 + r_y^2)dr$$

  where M is the total vessel mass, ρ is the density, and the integration is performed over the vessel volume. The reference frame is chosen so that the off-diagonal elements of the tensor vanish.
- The `shipedit` utility allows to calculate the inertia tensor from a mesh, assuming a homogeneous mass distribution.

void SetTrimScale (double) const

Sets the max. magnitude of the pitch trim control.

Synopsis:

```
void SetTrimScale (double scale) const
```

Parameters:

scale pitch trim scaling factor

Notes:

- If scale is set to zero (default) the vessel does not have a pitch trim control.

SetCameraOffset

Sets the camera position for internal (cockpit) view.

Synopsis:

```
void SetCameraOffset (const VECTOR3 &ofs) const
```

Parameters:

ofs camera offset in the vessel's local frame of reference [m,m,m]

Notes:

- Currently the camera direction in cockpit view is always the vessel's local +z axis (forward).

SetLiftCoeffFunc

Installs callback function for calculation of lift coefficient as a function of angle of attack.

Synopsis:

```
void SetLiftCoeffFunc (LiftCoeffFunc lcf) const
```

Parameters:

lcf — callback function pointer with the following interface:

```
double LiftCoeff (double aoa)
```

Notes:

- The callback function must be able to deal with aoa values in the range $-\pi$ ... π.
- If the function is not installed, the vessel is assumed not to produce any lift.

ParseScenarioLine

Processes an input line from a scenario file.

Synopsis:

```
void ParseScenarioLine (
     char *line,
     VESSELSTATUS *status) const
```

Parameters:

line — line to be interpreted

status — status parameter set

Notes:

- Normally, this function will be called from within the body of ovcLoadState to allow Orbiter to process any generic status parameters which are not processed by the module.

11.3. Current vessel status

GetStatus

Returns vessel's current status parameters.

Synopsis:

```
void GetStatus (VESSELSTATUS &status) const
```

Parameters:

status — struct receiving current vessel status

Notes:

- For a definition of VESSELSTATUS see Section 9.

DefSetState

Calls the default Orbiter vessel state initialisation with the specified status.

Synopsis:

```
void DefSetState (const VESSELSTATUS *status) const
```

Parameters:

status — vessel status parameters.

Notes:

- This function is most commonly used in ovcSetState to enable default state initialisation.

SaveDefaultState

Causes Orbiter to write default vessel parameters to a scenario file.

Synopsis:

```
void SaveDefaultState (FILEHANDLE scn) const
```

Parameters:

scn scenario file handle

Notes:

- This method should normally only be invoked from within ovcSaveState, to allow Orbiter to save its default vessel status parameters.
- If ovcSaveState is implemented but does not call SaveDefaultState, no default parameters are written to the scenario.

GetMass

Returns current (total) vessel mass. Equivalent to the oapiGetMass API function.

Synopsis:

```
double GetMass (void) const
```

Return value:

Current vessel mass [kg].

GetFuelMass

Returns the vessel's current fuel mass.

Synopsis:

```
double GetFuelMass (void) const
```

Return value:

Current fuel mass [kg]

GetEngineLevel

Returns the thrust level for an engine group.

Synopsis:

```
double GetEngineLevel (ENGINETYPE eng) const
```

Parameters:

eng engine group identifier

Return value:

thrust level (0..1)

Notes:

- For main engines, this does not include externally defined, module-controlled thrusters
- This function does not work for attitude thrusters.

SetEngineLevel

Sets the thrust level for an engine group.

Synopsis:

```
void SetEngineLevel (ENGINETYPE eng, double level) const
```

Parameters:

eng	engine group identifier
level	thrust level (0..1)

Notes:

- Main engine level –x is equivalent to retro engine level +x and vice versa.

IncEngineLevel

Increase or decrease the thrust level for an engine group.

Synopsis:

```
void IncEngineLevel (ENGINETYPE eng, double dlevel) const
```

Parameters:

eng	engine group identifier
dlevel	thrust increment

Notes:

- Use negative dlevel to decrease the engine's thrust level.
- Levels are clipped to valid range.

GetAttitudeMode

Returns the current attitude thruster mode.

Synopsis:

```
int GetAttitudeMode (void) const
```

Return value:

Current attitude mode (0=disabled or not available, 1=rotational, 2=linear)

SetAttitudeMode

Set the vessel's attitude thruster mode.

Synopsis:

```
bool SetAttitudeMode (int mode) const
```

Parameters:

mode	attitude mode (0=disable, 1=rotational, 2=linear)

Return value:

Error flag; *false* indicates error (requested mode not available)

GetAttitudeRotLevel

Returns the current thrust level for attitude thruster groups in rotational mode.

Synopsis:

```
void VESSEL::GetAttitudeRotLevel (VECTOR3 &th) const
```

Parameters:

th	vector containing thrust levels (-1 to 1)

Notes:

- The components of th are:
  th.x – attitude thrusters rotating around lateral axis
  th.y – attitude thrusters rotating around vertical axis
  th.z – attitude thrusters rotating around longitudinal axis
- To obtain the actual thrust force magnitudes [N], the absolute values must be multiplied with the max. attitude thrust (see GetMaxThrust())

See also:

VESSEL::GetMaxThrust(), VESSEL::GetAttitudeLinLevel(), VESSEL::GetAttitudeMode()

SetAttitudeRotLevel (1)

Set attitude thruster levels for rotation in all 3 axes.

Synopsis:

```
void SetAttitudeRotLevel (const VECTOR3 &th) const
```

Parameters:

th	attitude thruster levels for rotation around x,y,z axes

Notes:

- Thruster levels must be in the range [-1...1]
- This function works even if manual attitude mode is set to linear.

SetAttitudeRotLevel (2)

Set attitude thruster level for rotation around a single axis.

Synopsis:

```
void SetAttitudeRotLevel (int axis, double th) const
```

Parameters:

axis	rotation axis (0=x, 1=y, 2=z)
th	attitude thruster level

Notes:

- Thruster levels must be in the range [-1..1]
- This function works even if manual attitude mode is set to linear.

SetAttitudeLinLevel (1)

Set attitude thruster levels for linear translation in all 3 axes.

Synopsis:

```
void SetAttitudeLinLevel (const VECTOR3 &th) const
```

Parameters:

th	attitude thruster levels for translation along x,y,z

Notes:

- Thruster levels must be in the range [-1..1]
- This function works even if manual attitude mode is set to rotational.

SetAttitudeLinLevel (2)

Set attitude thruster level for linear translation along a single axis.

Synopsis:

```
void SetAttitudeLinLevel (int axis, double th) const
```

Parameters:

axis	translation axis (0=x, 1=y, 2=z)
th	attitude thruster level

Notes:

- Thruster levels must be in the range [-1..1]
- This function works even if manual attitude mode is set to rotational.

GetAttitudeLinLevel

Returns the current thrust level for attitude thrusters groups in linear mode.

Synopsis:

```
void VESSEL::GetAttitudeLinLevel (VECTOR3 &th) const
```

Parameters:

th vector containing thrust levels (-1 to 1)

Notes:

- The components of th are:
  th.x – attitude thrusters for lateral (sideways) translation
  th.y – attitude thrusters for vertical (up/down) translation
  th.z – attitude thrusters for longitudinal (forward/backward) translation
- To obtain the actual thrust force magnitudes [N], the absolute values must be multiplied with the max. attitude thrust (see GetMaxThrust())

See also:

VESSEL::GetMaxThrust(), VESSEL::GetAttitudeRotLevel(), VESSEL::GetAttitudeMode()

ActivateNavmode

Activates a navmode.

Synopsis:

```
bool ActivateNavmode (int mode)
```

Parameters:

mode navmode id to be activated.

Return value:

True if the specified navmode could be activated, false if not available or active already.

Notes:

- Navmodes are high-level navigation modes which involve e.g. the simultaneous and timed engagement of multiple attitude thrusters to get the vessel into a defined state. Some navmodes terminate automatically once the target state is reached (e.g. killrot), or they remain active until explicitly terminated (hlevel). Navmodes may also terminate if a second conflicting navmode is activated.
- For navmodes currently defined in Orbiter see the NAVMODE_xxx constants.

DeactivateNavmode

Deactivates a navmode.

Synopsis:

```
bool DeactivateNavmode (int mode)
```

Parameters:

mode navmode id to be deactivated.

Return value:

True if the specified navmode could be deactivated, false if not available or if deactivated already.

ToggleNavmode

Toggles a navmode on/off.

Synopsis:

```
bool ToggleNavmode (int mode)
```

Parameters:

mode navmode to be toggled.

Return value:

True if the navmode could be changed, false if it remains unchanged.

GetNavmodeState

Returns current state (on/off) of a navmode.

Synopsis:

```
bool GetNavmodeState (int mode)
```

Parameters:

mode navmode id to be checked.

Return value:

True if navmode is active, false otherwise.

SetFuelMass

Sets the vessel's current fuel mass [kg].

Synopsis:

```
void SetFuelMass (double m) const
```

Parameters:

m Current fuel mass [kg].

Notes:

- m must be between 0 and MaxFuelMass.
- This should NOT be used to implement normal fuel consumption by thrusters registered with Orbiter, since the Orbiter core takes care of this. Instead, this should be used for component modifications (e.g. stage separation), external thrust calculations, refuelling, etc.

11.4. State vectors

GetGlobalPos

Returns vessel's current position in the global reference frame.

Synopsis:

```
void GetGlobalPos (VECTOR3 &pos) const
```

Parameters:

pos: vector receiving position

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.0.
- Units are meters.
- Equivalent to `oapiGetGlobalPos(GetHandle(), &pos)`

GetGlobalVel

Returns vessel's current velocity in the global reference frame.

Synopsis:

```
void GetGlobalVel (VECTOR3 &vel) const
```

Parameters:

vel vector receiving velocity

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.0.
- Units are meters/second.
- Equivalent to `oapiGetGlobalVel (GetHandle(), &vel)`

GetRelativePos

Returns vessel's current position with respect to another object.

Synopsis:

```
void GetRelativePos (OBJHANDLE hRef, VECTOR3 &pos) const
```

Parameters:

hRef	reference object handle
pos	vector receiving position

Notes:

- Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.
- Equivalent to `oapiGetRelativePos (GetHandle(), hRef, &pos)`

GetRelativeVel

Returns vessel's current velocity relative to another object.

Synopsis:

```
void GetRelativeVel (OBJHANDLE hRef, VECTOR3 &pos) const
```

Parameters:

hRef	reference object handle
vel	vector receiving relative velocity

Notes:

- Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.
- Equivalent to oapiGetRelativeVel (GetHandle(), hRef, &vel)

GetEquPos

Returns vessel's current equatorial position (longitude, latitude and radius) with respect to the closest planet or moon.

Synopsis:

```
OBJHANDLE GetEquPos (
     double &longitude,
     double &latitude,
     double &radius) const
```

Parameters:

longitude	variable receiving longitude value [rad]
latitude	variable receiving latitude value [rad]
radius	variable receiving radius value [m]

Return value:

Handle to reference body to which the parameters refer. NULL indicates failure (no reference body available).

11.5. Orbital elements

Note: Calculating elements from state vectors is expensive. If possible, avoid calling the functions in this group at each frame (e.g. inside ovcTimestep). On the other hand, once any function in this group has been called, calling other functions during the *same* time step is not expensive.

GetGravityRef

Returns a handle to the main contributor of the gravity field at the vessel's current position.

Synopsis:

```
const OBJHANDLE GetGravityRef () const
```

Return value:

Handle to gravity reference object.

GetElements

Returns vessel's primary orbital elements w.r.t. dominant gravitational source.

Synopsis:

```
OBJHANDLE GetElements (ELEMENTS &el, double &mjd_ref) const
```

Parameters:

el	primary orbital elements (semi-major axis *a*, eccentricity *e*, inclination *i*, longitude of ascending node θ, longitude of periapsis ϖ, mean longitude at epoch *L*)
mjd_ref	reference epoch in MJD (Modified Julian Date) format

Return value:

Handle of reference object. NULL indicates failure (no elements available).

Notes:

- There are various ways to specify orbital elements. Note that here we use the *longitude* of the ascending node (not *anomaly* of the ascending node), and *longitude* of periapsis, and that the mean anomaly *L* refers to epoch (mjd_ref), not to date (so it should not change over time unless the orbit itself changes).

GetArgPer

Returns argument of periapsis.

Synopsis:

```
OBJHANDLE GetArgPer (double &arg) const
```

Parameters:

arg	argument of periapsis for current orbit [rad]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

GetSMi

Returns semi-minor axis.

Synopsis:

```
OBJHANDLE GetSMi (double &smi) const
```

Parameters:

smi	semi-minor axis for current orbit [m]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

GetApDist

Returns apoapsis distance.

Synopsis:

```
OBJHANDLE GetApDist (double &apdist) const
```

Parameters:

apdist apoapsis distance for current orbit [m]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

GetPeDist

Returns periapsis distance.

Synopsis:

```
OBJHANDLE GetPeDist (double &pedist) const
```

Parameters:

pedist periapsis distance for current orbit [m]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

11.6. Surface-relative parameters

GetSurfaceRef

Returns a handle to the closest planet or moon. This is the object to which all surface-relative parameters refer.

Synopsis:

```
const OBJHANDLE GetSurfaceRef () const;
```

Return value:

Handle to surface reference object (planet or moon)

GetAltitude

Returns altitude above closest planet/moon.

Synopsis:

```
double GetAltitude (void) const
```

Return value:

altitude [m]

GetHorizonAirspeedVector

Returns airspeed vector in local horizon coordinates.

Synopsis:

```
bool GetHorizonAirspeedVector (VECTOR3 &v) const
```

Parameters:

v variable receiving airspeed vector [m/s]

Return value:

false indicates error.

Notes:

- This function returns the airspeed vector in the reference frame of the local horizon. x = longitudinal component, y = vertical component, z = latitudinal component.

GetShipAirspeedVector

Returns airspeed vector in the vessel's local coordinates.

Synopsis:

```
bool GetShipAirspeedVector (VECTOR3 &v) const
```

Parameters:

v	variable receiving airspeed vector [m/s]

Return value:

false indicates error

Notes:

- This function returns the airspeed vector in local ship coordinates. x = lateral component, y = vertical component, z = longitudinal component.

GetAOA

Returns AOA (angle of attack). This is the pitch angle between the velocity vector and the vessel's longitudinal axis.

Synopsis:

```
double GetAOA (void) const
```

Return value:

angle of attack [rad]

GetSlipAngle

Returns the lateral (yaw) angle between the velocity vector and the vessel's longitudinal axis.

Synopsis:

```
double GetSlipAngle (void) const
```

Return value:

lateral slip angle [rad]

GetPitch

Returns pitch angle in local horizon frame.

Synopsis:

```
double GetPitch (void) const
```

Return value:

pitch angle [rad]

GetBank

Returns bank angle in local horizon frame.

Synopsis:

```
double GetBank (void) const
```

Return value:

bank angle [rad]

11.7. Transformations

ShiftCentreOfMass

Register a shift in the centre of mass after a structural change (e.g. stage separation)

Synopsis:

```
void ShiftCentreOfMass (const VECTOR3 &shift)
```

Parameters:

shift	CoM displacement vector.

Notes:

- This function should be called after a vessel has undergone a structural change which shifted the centre of mass, and which resulted in a change of the mesh component offsets of *-shift*. It will do two things:
  1. Translate the vessel's world reference point by +shift to compensate for the mesh offset shift.
  2. Drag the camera so that it centers at the new CoM (if in external mode tracking the concerned vessel).

GetRotationMatrix

Returns the vessel's current rotation matrix for transformations from the vessel's local frame of reference to the global (world) frame of reference.

Synopsis:

```
void GetRotationMatrix (MATRIX3 &R) const
```

Parameters:

R	rotation matrix

Notes:

- To transform a point $\mathbf{r}_{local}$ from local vessel coordinates to a global point $\mathbf{r}_{global}$, the following formula is used:
  $\mathbf{r}_{global} = \mathbf{R}\,\mathbf{r}_{local} + \mathbf{p}$,
  where **p** is the vessel's global position.
- This transformation can be directly performed by a call to `Local2Global`.

GlobalRot

Performs a rotation of a direction from the local vessel frame to the global frame.

Synopsis:

```
void GlobalRot (
     const VECTOR3 &rloc,
     VECTOR3 &rrot) const
```

Parameters:

rloc	point in local vessel coordinates (input)
rrot	rotated point (output)

Notes:

- This function is equivalent to multiplying rloc with the rotation matrix returned by `GetRotationMatrix`.
- Should be used to transform *directions*. To transform *points*, use `Local2Global`, which additionally adds the vessel's global position to the rotated point.

Local2Global

Performs a transformation from local vessel to global coordinates.

Synopsis:

```
void Local2Global (
     const VECTOR3 &local,
```

```
        VECTOR3 &global) const
```

Parameters:

local	point in local vessel coordinates (input)
global	transformed point in global coordinates (output)

Global2Local

Performs a transformation from global to local vessel coordinates.

Synopsis:

```
void Global2Local (
     const VECTOR3 &global,
     VECTOR3 &local) const
```

Parameters:

global	point in global coordinates (input)
local	transformed point in local vessel coordinates (output)

11.8. Atmospheric parameters

GetAtmPressure

Returns atmospheric pressure [Pascal] at current vessel position.

Synopsis:

```
double GetAtmPressure (void) const
```

Return value:

atmospheric pressure [Pa] at current vessel position.

Note:

- This function returns 0 if the vessel is outside all planetary atmospheric hulls, as defined by the planets' AtmAltLimit parameters.

GetAtmDensity

Returns atmospheric density [kg/m^3] at current vessel position.

Synopsis:

```
double GetAtmDensity (void) const
```

Return value:

atmospheric density [kg/m^3] at current vessel position.

Note:

- This function returns 0 if the vessel is outside all planetary atmospheric hulls, as defined by the planets' AtmAltLimit parameters.

11.9. Visual manipulation

ClearMeshes

Removes all previously declared meshes for the vessel's visual representation.

Synopsis:

```
void ClearMeshes () const
```

AddMesh (1)

Loads a new mesh from file and adds it to the vessel's visual representation.

Synopsis:

```
int AddMesh (
     const char *meshname,
     const VECTOR3 *ofs=0) const
```

Parameters:

meshname	mesh file name (without path and file extension) which must exist in the *Meshes* subdirectory.
ofs	optional pointer to a displacement vector which describes the offset (in meter) of the mesh origin against the vessel origin.

Return value:

mesh index

AddMesh (2)

This version adds a preloaded mesh to the vessel's visual representation.

Synopsis:

```
void AddMesh (MESHHANDLE hMesh, const VECTOR3 *ofs=0) const
```

Parameters:

hMesh	mesh handle
ofs	optional pointer to a displacement vector which describes the offset (in meter) of the mesh origin against the vessel origin.

Return value:

mesh index

See also:

oapiLoadMesh()

SetMeshVisibleInternal

Marks a mesh as visible from internal cockpit view.

Synopsis:

```
void SetMeshVisibleInternal (
     UINT meshidx,
     bool visible) const
```

Parameters:

meshidx	mesh index as returned by AddMesh()
visible	visibility flag

Notes:

- By default, a vessel is not rendered when the camera is in internal (cockpit) view. This function can be used to force rendering of some or all of the vessel's meshes.

AddExhaustRef

Adds a definition for rendering the exhaust from an engine.

Synopsis:

```
UINT AddExhaustRef (
     EXHAUSTTYPE exh,
     VECTOR3 &pos,
     double lscale = -1.0,
     double wscale = -1.0,
     VECTOR3 *dir = 0) const
```

Parameters:

exh	engine group identifier (main, retro, hover, custom)
pos	exhaust reference position (in local vessel coordinates)
lscale	longitudinal scaling factor
wscale	transversal scaling factor
dir	exhaust direction

Return value:

Exhaust identifier

Notes:

- The direction vector should be normalised to 1.
- For the standard engine types (main, retro, hover), default values are used for lscale, wscale and dir, if not supplied in the function call. Custom engines should always provide these values.
- For the standard engine types the scaling factors define the scaling for maximum thrust setting. For thrusts < 100% the scaling factors are adjusted accordingly. For custom engines, the scaling factors are used directly. The module is responsible for adjusting the scaling factors (via SetExhaustScales) to reflect changes in thrust.

DelExhaustRef

Deletes an exhaust render definition.

Synopsis:

```
void DelExhaustRef (EXHAUSTTYPE exh, WORD id) const
```

Parameters:

exh	engine group identifier (main, retro, hover, custom)
id	engine identifier, as returned by AddExhaustRef

ClearExhaustRefs

Deletes all exhaust render definitions.

Synopsis:

```
void ClearExhaustRefs (void)
```

SetExhaustScales

Sets the longitudinal and transversal scaling factors for exhaust rendering

Synopsis:

```
void SetExhaustScales (
     EXHAUSTTYPE exh,
     WORD id,
     double lscale,
     double wscale) const
```

Parameters:

exh	engine group identifier (main, retro, hover, custom)
id	engine identifier, as returned by AddExhaustRef
lscale	longitudinal scaling factor
wscale	transversal scaling factor

Notes:

- This function must be called for custom engines to reflect changes in thrust level. For standard engine types, this is done automatically.

AddAttExhaustRef

Adds an exhaust render definition for an attitude thruster.

Synopsis:

```
UINT AddAttExhaustRef (
     const VECTOR3 &pos,
     const VECTOR3 &dir,
     double wscale = 1.0,
     double lscale = 1.0) const
```

Parameters:

pos	exhaust reference position (in local vessel coordinates)
dir	exhaust direction (normalised)
wscale	exhaust render width scaling factor
lscale	exhaust render length scaling factor

Return value:

Attitude exhaust id.

Notes:

- This function only affects the exhaust rendering, not the physical parameters of the attitude engines.
- After creating an attitude thruster with AddAttExhaustRef, it must be assigned to one or more attitude modes with AddAttExhaustMode.

AddAttExhaustMode

Assign an attitude thruster to an attitude mode.

Synopsis:

```
void AddAttExhaustMode (
     UINT idx,
     ATTITUDEMODE mode,
     int axis,
     int dir) const
```

Parameters:

idx	attitude exhaust id, as returned by AddAttExhaustRef.
mode	ATTMODE_ROT or ATTMODE_LIN
axis	rotation/translation axis (0=x, 1=y, 2=z)
dir	rotation/translation direction (0 or 1)

Notes:

- An attitude thruster can be assigned to more than one mode (e.g. a rotational and a linear mode)
- Multiple attitude thrusters can be assigned to a single mode.
- The following attitude modes are available:

mode	**axis**	**dir**	**used for**
ATTMODE_ROT	0	0	pitch up
ATTMODE_ROT	0	1	pitch down
ATTMODE_ROT	1	0	yaw left
ATTMODE_ROT	1	1	yaw right
ATTMODE_ROT	2	0	roll right
ATTMODE_ROT	2	1	roll left
ATTMODE_LIN	0	0	move right
ATTMODE_LIN	0	1	move left
ATTMODE_LIN	1	0	move up
ATTMODE_LIN	1	1	move down
ATTMODE_LIN	2	0	move forward
ATTMODE_LIN	2	1	move back

ClearAttExhaustRefs

Deletes all attitude thruster exhaust render definitions.

Synopsis:

```
void ClearAttExhaustRefs (void) const
```

RegisterAnimation

Logs a request for calls to ovcAnimate, while the vessel's visual exists.

Synopsis:

```
void RegisterAnimation (void) const
```

Notes:

- This function allows to implement animation sequences in combination with the ovcAnimate callback function. After a call to RegisterAnimation, ovcAnimate is called at each time step, if the vessel's visual exists.
- Use UnregisterAnimation to stop further calls to ovcAnimate.
- Orbiter uses a reference counter to log animation requests. It calls ovcAnimate as long as counter > 0,
- If ovcAnimate is not implemented by the module, RegisterAnimation has no effect.

UnregisterAnimation

Unlogs an animation request.

Synopsis:

```
void UnregisterAnimation (void) const
```

Notes:

- This stops a request for animation callback calls from a previous RegisterAnimation.
- The call to UnregisterAnimation should not be placed in the body of ovcAnimate, since it may be lost if the vessel's visual doesn't exist.

RegisterAnimSequence

Register a "semi-automatic" animation sequence. (See Section 0, *Defining an animation sequence for a vessel*)

Synopsis:

```
UINT RegisterAnimSequence (double defstate) const
```

Parameters:

defstate animation state stored in the mesh.

Return value:

Animation sequence identifier.

Notes:

- Unlike `RegisterAnimation`/`UnregisterAnimation`, this function allows to create animation sequences which are processed by the Orbiter core, rather than manually by the module. The user only needs to define the components of the animation sequence once after creating the vessel, using `AddAnimComp()`, and can then manipulate the animation state via `SetAnimState()`.
- Each animation sequence is defined by its *state*, which has a value between 0 and 1. For example, for an animated landing gear operation state 0 may correspond to retracted gears, state 1 to fully deployed gears.
- defstate defines at which state the animation is stored in the mesh file.

AddAnimComp

Add a component to a previously registered animation sequence.

Synopsis:

```
bool AddAnimComp (UINT seq, ANIMCOMP *comp)
```

Parameters:

seq	sequence identifier, as returned by RegisterAnimSequence
comp	animation component description (see notes)

Return value:

false indicates failure.

Notes:

- ANIMCOMP is a structure defining the component's animation:

```
typedef struct {
        UINT *grp;
        UINT ngrp;
        double state0;
        double state1;
        MESHGROUP_TRANSFORM trans;
} ANIMCOMP;
```

  where:

grp:	array of group indices to be included in component
ngrp:	number of groups in the list
state0:	animation cutoff state 1
state1:	animation cutoff state 2
trans:	transformation description

- Note that in this case the *angle* or *shift* fields in MESHGROUP_TRANSFORM describe the *range* of animation, e.g. the angle over which a landing gear is rotated from fully retracted to fully deployed.
- state0 and state1 (0..1) allow to define the temporal endpoints of the component's animation within the sequence. For example, state0=0 and state1=1 perform the animation over the whole sequence animation, while state0=0 and state1=0.5 perform the animation over the first half of the sequence animation.

12. Class MFD

This class acts as an interface for user defined MFD (multi functional display) modes. It provides control over keyboard and mouse functions to manipulate the MFD mode, and allows the module to draw the MFD display. The MFD class is a pure virtual class. Each user-defined MFD mode requires the definition of a specialised class derived from MFD. An example for a generic MFD mode implemented as a plugin module can be found in orbitersdk\samples\CustomMFD.

Public member functions

12.1. Construction/creation

MFD

Constructor. Creates a new MFD.

Synopsis:

```
MFD (DWORD w, DWORD h, VESSEL *vessel)
```

Parameters:

w	width of the MFD display (pixel)
h	height of the MFD display (pixel)
vessel	pointer to VESSEL interface associated with the MFD.

Notes:

- MFD is a pure virtual function, so it can't be instantiated directly. It is used as a base class for specialised MFD modes.
- New MFD modes are registered by a call to oapiRegisterMFDMode. Whenever the new mode is selected by the user, Orbiter sends a OAPI_MSG_MFD_OPENED signal to the message handler, to which the module should respond by creating the MFD mode and returning a pointer to it. Orbiter will automatically destroy the MFD mode when it is turned off.

12.2. Display repaint

Update

Called when the MFD needs to update its display.

Synopsis:

```
virtual void Update (HDC hDC) = 0
```

Parameters:

hDC Windows device context for drawing on the MFD display surface.

Notes:

- The frequency at which this function is called corresponds to the "MFD refresh rate" setting in Orbiter's parameter settings.
- This function must be overwritten by derived classes.

SelectDefaultFont

Selects a predefined MFD font into the device context.

Synopsis:

```
HFONT SelectDefaultFont (HDC hDC, DWORD i) const
```

Parameters:

hDC Windows device context
i font index

Return value:

Windows font handle

Notes:

- Currently supported are font indices 0-2, where
  0 = standard MFD font (Courier, fixed pitch)
  1 = small font (Arial, variable pitch)
  2 = small font, rotated 90 degrees (Arial, variable pitch)
- In principle, an MFD mode may create its own fonts using the standard Windows *CreateFont* function, but using the predefined fonts is preferred to provide a consistent MFD look.
- Default fonts are scaled automatically according to the MFD display size.

SelectDefaultPen

Selects a predefined pen into the device context.

Synopsis:

```
HPEN SelectDefaultPen (HDC hDC, DWORD i) const
```

Parameters:

hDC Windows device context
i pen index

Return value:

Windows pen handle

Notes:

- Currently supported are pen indices 0-5, where
  0 = solid, HUD display colour
  1 = solid, light green
  2 = solid, medium green
  3 = solid, medium yellow
  4 = solid, dark yellow
  5 = solid, medium grey
- In principle, an MFD mode may create its own pen resources using the standard Windows *CreatePen* function, but using predefined pens is preferred to provide a consistent MFD look.

ButtonLabel

Return the label for the specified MFD button.

Synopsis:

```
virtual char *ButtonLabel (int bt)
```

Parameters:

bt button identifier

Return value:

The function should return a 0-terminated character string of up to 3 characters, or NULL if the button is unlabelled.

ButtonMenu

Defines a list of short descriptions for the various MFD mode button/key functions.

Synopsis:

```
virtual int  ButtonMenu (const MFDBUTTONMENU **menu) const
```

Parameters:

menu on return this should point to an array of button menu items. (see notes)

Return value:

number of items in the list

Notes:

- The definition of the MFDBUTTONMENU struct is:
  ```
  typedef struct {
    const char *line1, *line2;
    char selchar;
  } MFDBUTTONMENU;
  ```
  containing up to 2 lines of short description, and the keyboard key to trigger the function.
- Each line should contain no more than 16 characters, to fit into the MFD display.
- If the menu item only uses one line, then line2 should be set to NULL.
- menu==0 is valid and indicates that the caller only requires the number of items, not the actual list.
- A typical implementation would be

```
int MyMFD::ButtonMenu (const MFDBUTTONMENU **menu) const
{
  static const MFDBUTTONMENU mnu[2] = {
    {"Select target", 0, 'T'},
    {"Select orbit", "reference", 'R'}
  };
  if (menu) *menu = mnu;
  return 2;
}
```

12.3. Input

ConsumeKeyBuffered

MFD keyboard handler for buffered keys.

Synopsis:

```
virtual bool ConsumeKeyBuffered (DWORD key)
```

Parameters:

key	key code (see OAPI_KEY_xxx constants in orbitersdk.h)

Return value:

The function should return true if it recognises and processes the key, false otherwise.

ConsumeKeyImmediate

MFD keyboard handler for immediate (unbuffered) keys.

Synopsis:

```
virtual bool ConsumeKeyImmediate (char *kstate)
```

Parameters:

kstate:	keyboard state.

Return value:

The function should return true only if it wants to inhibit Orbiter's default immediate key handler for this time step completely.

Notes:

- The state of single keys can be queried by the KEYDOWN macro.
- The immediate key handler is useful where an action should take place *while a key is pressed*.

ConsumeButton

MFD button handler. This function is called when the user performs a mouse click on a panel button associated with the MFD.

Synopsis:

```
virtual bool ConsumeButton (int bt, int event)
```

Parameters:

bt	button identifier.
event	mouse event (see PANEL_MOUSE_xxx constants in orbitersdk.h)

Return value:

The function should return true if it processes the button event, false otherwise.

Notes:

- This function is invoked as a response to a call to `oapiProcessMFDButton` in a vessel module.
- Typically, `ConsumeButton` will call `ConsumeKeyBuffered` or `ConsumeKeyImmediate` to emulate a keyboard event.

12.4. Load/save state

WriteStatus

Called when the MFD should write its status to a scenario file.

Synopsis:

```
virtual void WriteStatus (FILEHANDLE scn) const
```

Parameters:

scn scenario file handle (write only)

Notes:

- Use the oapiWriteScenario_xxx functions to write MFD status parameters to the scenario.
- The default behaviour is to do nothing. MFD modes which need to save status parameters should overwrite this function.

ReadStatus

Called when the MFD should read its status from a scenario file.

Synopsis:

```
virtual void ReadStatus (FILEHANDLE scn)
```

Parameters:

scn scenario file handle (read only)

Notes:

- Use a loop with oapiReadScenario_nextline to read MFD status parameters from the scenario.
- The default behaviour is to do nothing. MFD modes which need to read status parameters should overwrite this function.

StoreStatus

Called before destruction of the MFD mode, to allow the mode to save its status to static memory.

Synopsis:

```
virtual void StoreStatus (void) const
```

Notes:

- This function is called before an MFD mode is destroyed (either because the MFD switches to a different mode, or because the MFD itself is destroyed). It allows the MFD to back up its status parameters, so it can restore its last status when it is created next time.
- Since the MFD mode instance is about to be destroyed, status parameters should be backed up either in static data members, or outside the class instance.
- In principle this function could be implemented by opening a file and calling WriteStatus with the file handle. However for performance reasons file I/O should be avoided in this function.
- The default behaviour is to do nothing. MFD modes which need to save status parameters should overwrite this function.

RecallStatus

Called after creation of the MFD mode, to allow the mode to restore its status from the last save.

Synopsis:

```
virtual void RecallStatus (void)
```

Notes:

- This is the counterpart to the StoreStatus function. It should be implemented if and only if StoreStatus is implemented.

13. Class GraphMFD

This class is derived from MFD and provides a template for MFD modes containing 2D graphs. An example is the ascent profile recorder in the samples\CustomMFD folder.

13.1. Construction/creation

GraphMFD

Constructor. Creates a new GraphMFD.

Synopsis:

```
GraphMFD (DWORD w, DWORD h, VESSEL *vessel)
```

Parameters:

w	width of the MFD display (pixel)
h	height of the MFD display (pixel)
vessel	pointer to VESSEL interface associated with the MFD

13.2. Graph/plot management

AddGraph

Adds a new graph to the MFD.

Synopsis:

```
int AddGraph (void)
```

Return value:

graph identifier

Notes:

- This function allocates data for a new graph. To display plots in the new graph, one or more calls to AddPlot are required.

AddPlot

Adds a plot to an existing graph.

Synopsis:

```
void AddPlot (
     int g,
     float *absc,
     float *data,
     int ndata,
     int col,
     int *ofs = 0)
```

Parameters:

g	graph identifier
absc	pointer to array containing the abscissa (x-axis) values.
data	pointer to array containing the data (y-axis) values.
ndata	number of data points
col	plot colour index
ofs	pointer to data offset (optional)

Notes:

- Data arrays are not copied, so they should not be deleted after the call to AddPlot.
- col is used as an index to select a pen for the plot using the SelectDefaultPen function. Valid range is the same as for SelectDefaultPen.
- If defined, *ofs is the index of the first plot value in the data array. The plot is drawn using the points *ofs to ndata-1, followed by points 0 to *ofs-1. This allows to define continuously updated plots without having to copy blocks of data within the arrays.

SetRange

Sets a fixed range for the x or y axis of a graph.

Synopsis:

```
void SetRange (int g, int axis, float rmin, float rmax)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)
rmin	minimum value
rmax	maximum value

Notes:

- The range applies to all plots in the graph.

SetAutoRange

Allows the graph to set its range automatically according to the range of the plots.

Synopsis:

```
void SetAutoRange (int g, int axis, int p = -1)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)
p	plot identifier (-1=all)

Notes:

- If $p \geq 0$, then p specifies the plot used for determining the graph range. If p = -1, then all of the graph's plots are used to determine the range.

FindRange

Determines the range of an array of data.

Synopsis:

```
void FindRange (
     float *d,
     int ndata,
     float &dmin,
     float &dmax) const
```

Parameters:

d	data array
ndata	number of data
dmin	minimum value on return
dmax	maximum value on return

SetAxisTitle

Sets the title for a given graph and axis.

Synopsis:

```
void SetAxisTitle (int g, int axis, char *title)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)
title	axis title

Notes:

- The MFD may append an extension of the form "x <scale>" to the title, where <scale> is a scaling factor applied to the tick labels of the axis. It is therefore a good idea to finish the title with the units applicable to the data of this axis, so that for example a title "Altitude: km" may become "Altitude: km x 1000".

SetAutoTicks

Calculates tick intervals for a given graph and axis.

Synopsis:

```
void SetAutoTicks (int g, int axis)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)

Notes:

- This function is called from within SetRange and normally doesn't need to be called explicitly by derived classes.

Plot

Displays a graph.

Synopsis:

```
void Plot (
     HDC hDC,
     int g,
     int h0,
     int h1,
     const char *title = 0)
```

Parameters:

hDC	Windows device context
g	graph identifier
h0	upper boundary of plot area (pixel)
h1	lower boundary of plot area (pixel)
title	graph title

Notes:

- This function should be called from Update to paint the graph(s) into the provided device context.

14. Plugin callback function reference

This is a list of callback functions which Orbiter will call for all activated *plugin modules*. (i.e. DLLs in the Modules\Plugin subdirectory which were activated by the user via the Launchpad dialog). Plugin callback functions use an *opc* ("orbiter plugin callback") prefix.

opcDLLInit

Called after the DLL is loaded by Orbiter, before the simulation window is opened. DLLs are loaded either during the program start, or when the user activates a DLL in the *Modules* tab of the launchpad dialog.

Synopsis:

```
DLLCLBK void opcDLLInit (HINSTANCE hDLL)
```

Parameters:

hDLL	DLL module handle

opcDLLExit

Called before the DLL is unloaded by Orbiter, after the simulation window has closed. DLLs are unloaded either when Orbiter exits, or when the user deactivates a DLL in the *Modules* tab of the launchpad dialog.

Synopsis:

```
DLLCLBK void opcDLLExit (HINSTANCE hDLL)
```

Parameters:

hDLL	DLL module handle

opcOpenRenderViewport

Called after the simulation window has been opened. The DLL should use this function for initialisations which depend on the size of the render window. The size remains valid until the opcCloseRenderViewport method is called. Note that for windowed modes the width and height parameters may be smaller than the user-defined window size, to accommodate window borders and title line.

Synopsis:

```
DLLCLBK void opcOpenRenderViewport (
     HWND renderWnd,
     DWORD width,
     DWORD height,
     BOOL fullscreen)
```

Parameters:

renderWnd	render window handle
width	width of the render viewport (pixel)
height	height of the render viewport (pixel)
fullscreen	TRUE if a fullscreen video mode is used, FALSE for a windowed mode

opcCloseRenderViewport

Called after the simulation window has been closed.

Synopsis:

```
DLLCLBK void opcCloseRenderViewport (void)
```

opcTimestep

Called at each time step of the simulation. Note that this is not exactly the same as an animation frame, because rendering may continue during a simulation pause if the camera is movable during pause.

Synopsis:

```
DLLCKBK void opcTimestep (
     double SimT,
     double SimDT,
     double mjd)
```

Parameters:

SimT	elapsed simulation time since simulation start (seconds)
SimDT	time interval since last time step (seconds)
mjd	simulation universal time in MJD (modified Julian date) format.

opcFocusChanged

Called when input focus (keyboard and joystick control) is switched to a new vessel (for example as a result of a call to oapiSetFocus).

Synopsis:

```
DLLCLBK opcFocusChanged (
     OBJHANDLE new_focus,
     OBJHANDLE old_focus)
```

Parameters:

new_focus	handle of vessel receiving the input focus
old_focus	handle of vessel losing focus

Notes:

Currently only objects of type “vessel” can receive the input focus. This may change in future versions.

opcTimeAccChanged

Called when the simulation time acceleration factor changes.

Synopsis:

```
DLLCLBK void opcTimeAccChanged (
     double nWarp,
     double oWarp)
```

Parameters:

nWarp	new time acceleration factor
oWarp	old time acceleration factor

15. Vessel callback function reference

This is a list of callback functions for *vessel modules* (i.e. modules referenced by the Module entry in vessel class configuration files). Vessel callback functions use an *ovc* (“orbiter vessel callback”) prefix.

ovcInit

Called during vessel creation. A vessel module *must* define this function in order to create an instance of the VESSEL interface or a derived class.

Synopsis:

```
DLLCLBK VESSEL *ovcInit (
     OBJHANDLE hVessel,
     int flightmodel)
```

Parameters:

hVessel	handle identfying the newly created vessel.
flightmodel	level of flight model realism (0=simple, 1=realistic)

Return value:

Module-generated instance of VESSEL or a derived class.

Notes:

A typical implementation will look like this:

```
class MyVessel: public VESSEL
{
   ...
}

DLLCLBK VESSEL *ovcInit (OBJHANDLE hVessel, int flightmodel)
{
   return new MyVessel(hVessel, flightmodel);
}
```

ovcExit

Called before killing the vessel. Should be used for cleanup operations (memory deallocation etc.) and for deallocating the VESSEL interface.

Synopsis:

```
DLLCLBK void ovcExit (VESSEL *vessel)
```

Parameters:

vessel	vessel interface

ovcSetClassCaps

Called during vessel initialisation. This allows the module to define vessel class capabilities, such as mass, size, aerodynamic specs, thruster ratings, etc.

Synopsis:

```
DLLCLBK void ovcSetClassCaps (
     VESSEL *vessel,
     FILEHANDLE cfg)
```

Parameters:

vessel	vessel interface
cfg	handle for the vessel class configuration file.

Notes:

- This function should only set general parameters (like maximum fuel mass), not the current state parameters for a specific ship (like current fuel mass).
- Generic parameters directly defined in the vessel class cfg file (e.g. *MaxFuel*) override values set in ovcSetClassCaps. This allows to manipulate values without need to recompile the module.
- The cfg file handle allows to read nonstandard parameters from the class file.

ovcSetState

Called at vessel creation to allow initialisation of the initial state.

Synopsis:

```
DLLCLBK void ovcSetState (
     VESSEL *vessel,
     const VESSELSTATUS *status)
```

Parameters:

vessel	vessel interface
status	vessel state parameters

Notes:

- This function is called after `ovcSetClassCaps`.
- If this function is not defined, Orbiter will perform default state initialisations.
- To perform Orbiter's default initialisation from within ovcSetState, call `vessel->DefSetState (status)`

ovcLoadState

Called when the vessel must read its initial status from a scenario file.

Synopsis:

```
DLLCLBK void ovcLoadState (
     VESSEL *vessel,
     FILEHANDLE scn,
     VESSELSTATUS *def_vs)
```

Parameters:

vessel	vessel interface
scn	scenario file handle
def_vs	set of generic vessel parameters

Notes:

- This callback function is provided to allow the module to read non-standard parameters from the scenario file.
- The function should define a loop which parses lines from the scenario file via oapiReadScenario_nextline.
- Any lines which the module parser does not recognise should be forwarded to Orbiter's default scenario parser via VESSEL::ParseScenarioLine, to allow the processing of generic options.

- Alternatively, the module parser may intercept generic parameters and directly write values into the generic set def_vs (dangerous!)
- A typical parser may look like this:

```
DLLCLBK void ovcLoadState (VESSEL *vessel, FILEHANDLE scn,
    VESSELSTATUS *def_vs)
{
    char *line;
    int my_value;

    while (oapiReadScenario_nextline (scn, line)) {
        if (!strnicmp (line, "my_option", 9)) {
            sscanf (line+9, "%d", &my_value);
        } else if (...) { // more items
            ...
        } else { // anything not recognised is passed to Orbiter
            vessel->ParseScenarioLine (line, def_vs);
        }
    }
}
```

ovcSaveState

Called when a vessel needs to save its current status to a scenario file.

Synopsis:

```
DLLCLBK void ovcSaveState (
    VESSEL *vessel,
    FILEHANDLE scn)
```

Parameters:

vessel	vessel interface
scn	scenario file handle

Notes:

- This function only needs to be implemented if the vessel must save non-standard parameters. Otherwise Orbiter invokes a default parameter save.
- To allow Orbiter to save its default vessel parameters, use VESSEL::SaveDefaultState.
- To write custom parameters to the scenario file, use the oapiWriteLine method.

ovcVisualCreated

Called after a the visual representation of a vessel has been created.

Synopsis:

```
DLLCLBK void ovcVisualCreated (
    VESSEL *vessel,
    VISHANDLE vis,
    int refcount)
```

Parameters:

vessel	vessel interface
vis	handle for the newly created visual
refcount	visual reference count

Notes:

- The logical interface to a vessel exists as long as the vessel is present in the simulation. However, the visual interface exists only when the vessel is within visual range of the camera. Orbiter creates and destroys visuals as required. This enhances simulation performance in the presence of a large number of objects in the simulation.
- Whenever Orbiter creates a vessel's visual it reverts to its initial configuration (e.g. as defined in the mesh file). The module can use this function to update the visual to the current state, wherever dynamic changes are required.

- More than one visual representation of an object may exist. The refcount parameter defines how many visual interfaces to the object exist.

ovcVisualDestroyed

Called before the visual representation of a vessel is destroyed.

Synopsis:

```
DLLCLBK void ovcVisualDestroyed (
     VESSEL *vessel,
     VISHANDLE vis,
     int refcount)
```

Parameters:

vessel	vessel interface
vis	handle for the visual to be destroyed
refcount	visual reference count

Notes:

- Orbiter calls this function before it destroys the vessel's visual representation, e.g. when it moves out of the visual range of the current camera.
- The (logical) vessel may still exist, but it is no longer rendered.

ovcTimestep

Called at each simulation time step *before* the vessel updates its position and velocity.

Synopsis:

```
DLLCLBK void ovcTimestep (VESSEL *vessel, double simt)
```

Parameters:

vessel	vessel interface
simt	simulation up time (seconds since simulation start)

Notes:

- This function, if implemented, is called at each frame for each instance of this vessel class, and is therefore time-critical. Avoid any unnecessary calculations here which may degrade performance.

ovcNavmode

Called at activation/deactivation of a navmode (see also VESSEL::ActivateNavmode)

Synopsis:

```
DLLCLBK void ovcNavmode (
     VESSEL *vessel,
     int mode,
     bool active)
```

Parameters:

vessel	vessel interface
mode	navmode constant (see section 10)
active	true for activation, false for deactivation.

ovcHUDmode

Called after a change of the vessel's HUD (head up display) mode.

Synopsis:

```
DLLCLBK void ovcHUDmode (VESSEL *vessel, int mode)
```

Parameters:

vessel	vessel interface
mode	new HUD mode

Notes:

- For currently supported HUD modes see `HUD_xxx` constants in section 10.
- mode `HUD_NONE` indicates that the HUD has been turned off.

ovcMFDmode

Called after the display mode of one of the MFDs (multifunctional displays) has changed.

Synopsis:

```
DLLCLBK void ovcMFDmode (VESSEL *vessel, int mfd, int mode)
```

Parameters:

vessel	vessel interface
mfd	MFD identifier (see Section 10).
mode	new MFD mode (see Section 10).

ovcAnimate

Called at each simulation time step if the module has registered an animation request and if the vessel's visual exists.

Synopsis:

```
DLLCLBK void ovcAnimate (VESSEL *vessel, double simt)
```

Parameters:

vessel	vessel interface
simt	simulation up time (seconds since simulation start)

Notes:

- This callback allows the module to animate the vessel's visual representation (moving undercarriage, cargo bay doors, etc.)
- It is only called as long as the vessel has registered an animation (between matching VESSEL::RegisterAnimation and VESSEL::UnregisterAnimation calls) and if the vessel's visual exists.
- The UnregisterAnimation call should not be placed within the body of ovcAnimate, since it would be lost if the vessel's visual doesn't exist. This should rather be placed in ovcTimestep.

ovcConsumeKey

Keyboard handler. Called at each simulation time step. This callback function allows the installation of a custom keyboard interface for the vessel.

Synopsis:

```
DLLCLBK int ovcConsumeKey (
     VESSEL *vessel,
     const char *keystate)
```

Parameters:

vessel	vessel interface
keystate	keyboard state

Return value:

A nonzero return value will completely disable default processing of the key state for the current time step. To disable the default processing of selected keys only, use the RESETKEY macro (see orbitersdk.h) and return 0.

Notes:

- The keystate contains the current keyboard state. Use the KEYDOWN macro in combination with the key identifiers as defined in orbitersdk.h (OAPI_KEY_xxx) to check for particular keys being pressed. Example:

```
if (KEYDOWN (keystate, OAPI_KEY_F10)) {
  // perform action
  RESETKEY (keystate, OAPI_KEY_F10);
  // optional: prevent default processing of the key
}
```

- This function should be used where a key *state*, rather than a key *event* is required, for example when engaging thrusters or similar. To test for key events (key pressed, key released) use ovcConsumeBufferedKey instead.

ovcConsumeBufferedKey

This callback function notifies the module of a buffered key event (key pressed or key released).

Synopsis:

```
DLLCLBK int ovcConsumeBufferedKey (
    VESSEL *vessel,
    DWORD key,
    bool down,
    char *kstate)
```

Parameters:

vessel	vessel interface
key	key scan code (see OAPI_KEY_xxx constants in orbitersdk.h)
down	true if key was pressed, false if key was released
kstate	current keyboard state

Return value:

The function should return 1 if Orbiter's default processing of the key should be skipped, 0 otherwise.

Notes:

- The key state (kstate) can be used to test for key modifiers (Shift, Ctrl, etc.). The KEYMOD_xxx macros defined in orbitersdk.h are useful for this purpose.
- This function may be called repeatedly during a single frame, if multiple key events have occurred in the last time step.

ovcLoadPanel

Called when Orbiter needs to load a custom instrument panel from the module.

Synopsis:

```
DLLCLBK bool ovcLoadPanel (VESSEL *vessel, int id)
```

Parameters:

vessel	vessel interface
id	panel identifier

Return value:

false indicates failure.

Notes:

- In the body of this function the module should define the panel background bitmap, and panel capabilities, e.g. the position of MFDs and other instruments, active areas (mouse hotspots) etc.
- A vessel which implements panels must at least support panel id 0 (the main panel. If any panels register neighbour panels (see oapiSetPanelNeighbours), all the neighbours must be supported, too.
- See also: oapiRegisterPanelBackground, oapiRegisterPanelArea, oapiRegisterMFD.

ovcPanelMouseEvent

Called when a previously registered panel area receives a mouse button event.

Synopsis:

```
DLLCLBK bool ovcPanelMouseEvent (
     VESSEL *vessel,
     int id,
     int event,
     int mx,
     int my)
```

Parameters:

vessel	vessel interface
id	panel area identifier
event	mouse event (see PANEL_MOUSE_xxx constants in orbitersdk.h)
mx, my	relative mouse position in area at event

Return value:

The function should return *true* if it processes the event, false otherwise.

Notes:

- Mouse events are only sent for areas which requested notification during definition (see oapiRegisterPanelArea).

ovcPanelRedrawEvent

Called when a panel area receives a redraw event.

Synopsis:

```
DLLCLBK bool ovcPanelRedrawEvent (
     VESSEL *vessel,
     int id,
     int event,
     SURFHANDLE surf)
```

Parameters:

vessel	vessel interface
id	panel area identifier
event	redraw event (see PANEL_REDRAW_xxx constants in orbitersdk.h)
surf	area surface handle.

Return value:

The function should return *true* if it processes the event, false otherwise.

Notes:

- This callback function is only called for areas which were not registered with the PANEL_REDRAW_NEVER flag.
- All redrawable panel areas receive a PANEL_REDRAW_INIT redraw notification when the panel is created, in addition to any registered redraw notification events.
- The surface handle surf contains either the *current area state*, or the *area background*, depending on the flags passed during area registration.
- The surface handle may be used for blitting operations, or to receive a Windows device context (DC) for Windows-style redrawing operations.

See also:

oapiGetDC, oapiReleaseDC, oapiTriggerPanelRedrawArea

16. Planet callback function reference

This is a list of callback functions Orbiter will call for all *planet modules* (i.e. modules referenced by the *Module* entry in the configuration files of planets or moons). See also the Vsop87 entry in the "Standard Orbiter modules" section below.

In the following <Planet> is a placeholder for the planet's or moon's name as defined in its configuration file (case-sensitive!)

<Planet>_SetPrecision

Define the relative error for the calculations for <Planet>.

Synopsis:

```
DLLCLBK int <Planet>_SetPrecision (double prec)
```

Parameters:

prec	module-specific

Return value:

not used by Orbiter

Notes:

- Orbiter calls this function at the start of each simulation with the value of the *ErrorLimit* entry of the planet's configuration file. The module can use this to set its calculation precision.
- If the *ErrorLimit* entry is not defined in the cfg file, then <Planet>_SetPrecision will not be called, so the module should initialise some default precision.
- It is up to the module how to interpret the passed precision value, but by convention prec should specify the relative error for position and velocity calculations.
- This function is optional. If the module doesn't define it, Orbiter will ignore the *ErrorLimit* entry in the cfg file.

<Planet>_EclSphData

Calculate ecliptic positions and velocities in spherical coordinates. Reference frame is ecliptic and equinox of J2000. For planets (i.e. objects defined as "Planet" in the solar system cfg file) heliocentric coordinates should be calculated. For moons (i.e. objects defined as "Moon" in the solar system cfg file) coordinates w.r.t. the moon's reference planet should be calculated, e.g. geocentric for Earth's moon.

Synopsis:

```
DLLCLBK int <Planet>_EclSphData (double mjd, double *ret)
```

Parameters:

mjd	date in MJD format (MJD = JD-2400000.5)
ret	array of results which the function should calculate as follows: ret[0] = longitude [rad] ret[1] = latitude [rad] ret[2] = radius [AU] ret[3] = velocity in longitude [rad/s] ret[4] = velocity in latitude [rad/s] ret[5] = radial velocity [AU/s]

Return value:

Error code (not used)

Notes:

- The function should calculate the values for ret in the J2000 ecliptic frame, but Orbiter's precision requirements are not very high, so the ecliptic of a different epoch (or the ecliptic of date) is probably ok.
- Orbiter only calls this function directly to calculate positions at times other than the current simulation time (e.g. for trajectory predictions). Otherwise it calls <Planet>_CurrentData (see below).

<Planet>_CurrentData

This function is called by Orbiter at each frame to update planet positions and velocities. Therefore the implementation can make use of interpolation methods to increase the efficiency of the calculation.

Synopsis:

```
DLLCLBK int <Planet>_CurrentData (
    double simt,
    double *ret)
```

Parameters:

simt	Time (in seconds) since simulation start
ret	results (as in <Planet>_EclSphData)

Return value:

not used

Notes:

- Orbiter passes simt (simulation time in seconds) rather than mjd to this function to allow more precise calculation of the interpolation point.
- The simplest way to implement this function is as `return <Planet>_EclSphData (oapiTime2MJD (simt), ret);` However this is not recommended. Instead the function should sample the planet data in appropriate intervals and use an interpolation scheme to calculate the data for a given time. This is more efficient and helps smoothing rounding errors in the full updates.
- This function is called at every frame by Orbiter and is therefore extremely time-critical. As a performance target, the execution of this function for *all* planets should take < 10 milliseconds on a low-end machine.
- The sampling times for full position calculations should be staggered for different planets, so that not all full updates occur at the same frame.

17. API function reference

This is the reference list for the Orbiter API functions which can be used by modules to obtain and set simulation parameters from the Orbiter kernel. See index for alphabetical listing.

17.1. Obtaining object handles

oapiGetObjectByName

Retrieve the handle for an object from its name. Objects may be vessels, orbital stations, planets, moons or suns. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:

```
OBJHANDLE oapiGetObjectByName (char *name)
```

Parameters:

name	object name (not case-sensitive)

Return value:

object handle. (NULL indicates that the object does not exist)

oapiGetObjectByIndex

Retrieve the handle for an object from its index. This is useful to construct loops over a series of objects. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:

```
OBJHANDLE oapiGetObjectByIndex (int index)
```

Parameters:

index object index (>= 0)

Return value:

object handle. (NULL indicates that the object does not exist)

Notes:

0 <= index < *oapiGetObjectCount()* is required. The function does not perform a range check!

oapiGetObjectCount

Returns the number of objects currently present in the simulation.

Synopsis:

```
DWORD oapiGetObjectCount (void)
```

Return value:

object count

oapiGetVesselByName

Retrieve the handle for a vessel from its name. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:

```
OBJHANDLE oapiGetVesselByName (char *name)
```

Parameters:

name object name (not case-sensitive)

Return value:

object handle. (NULL indicates that the vessel does not exist)

oapiGetVesselByIndex

Retrieve the handle for a vessel from its index. This is useful to construct loops over a series of vessels. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:

```
OBJHANDLE oapiGetVesselByIndex (int index)
```

Parameters:

index object index (>= 0)

Return value:

vessel handle. (NULL indicates that the vessel does not exist)

Notes:

0 <= index < *oapiGetVesselCount()* is required. The function does not perform a range check!

oapiGetVesselCount

Returns the number of vessels currently present in the simulation.

Synopsis:
```
DWORD oapiGetVesselCount (void)
```

Return value:
vessel count

oapiGetStationByName
Retrieves the handle of an orbital station from its name.

Synopsis:
```
OBJHANDLE oapiGetStationByName (char *name)
```

Parameters:
name station name (not case-sensitive)

Return value:
object handle. (NULL indicates that the station does not exist)

oapiGetStationByIndex
Retrieves the handle of an orbital station from its list index.

Synopsis:
```
OBJHANDLE oapiGetStationByIndex (int index)
```

Parameters:
index object index (>= 0)

Return value:
object handle. (NULL indicates that the station does not exist)

oapiGetStationCount
Returns the number of stations currently in the simulation.

Synopsis:
```
DWORD oapiGetStationCount (void)
```

Return value:
station count

oapiGetGbodyByName
Retrieves the handle of a “massive” object (a gravitational field source: sun, planet or moon) from its name.

Synopsis:
```
OBJHANDLE oapiGetGbodyByName (char *name)
```

Parameters:
name object name (not case-sensitive)

Return value:
object handle. (NULL indicates that the object does not exist)

oapiGetGbodyByIndex
Retrieves the handle of a massive object from its list index.

Synopsis:
```
OBJHANDLE oapiGetGbodyByIndex (int index)
```

Parameters:

index object index (>= 0)

Return value:
object handle. (NULL indicates that the object does not exist)

oapiGetGbodyCount
Returns the number of massive objects (suns, planets and moons) currently in the simulation.

Synopsis:
```
DWORD oapiGetGbodyCount ()
```

Return value:
Number of objects

oapiGetObjectName
Returns the name of an object.

Synopsis:
```
void oapiGetObjectName (
     OBJHANDLE hObj,
     char *name,
     int n)
```

Parameters:
hObj object handle
name pointer to character array to receive object name
n length of string buffer

Notes:
name must be allocated to at least size *n* by the calling function.
If the string buffer is not long enough to hold the object name, the name is truncated.

oapiGetFocusObject
Retrieve the handle for the current focus object. The focus object is the user-controlled vessel which receives keyboard and joystick input.

Synopsis:
```
OBJHANDLE oapiGetFocusObject (void)
```

Return value:
focus object handle. This is guaranteed to exist during the simulation (between *opcOpenRenderViewport* and *opcCloseRenderViewport*)

Notes:
Currently the focus object is guaranteed to be a vessel. This may change in future versions.

oapiSetFocusObject
Switches the input focus to a different vessel object.

Synopsis:
```
OBJHANDLE oapiGetFocus (OBJHANDLE hVessel)
```

Parameters:
hVessel handle of vessel to receive the focus

Return value:
handle of vessel losing focus, or NULL if focus did not change

Notes:

hVessel must refer to a vessel object. Trying to set the focus to a different object type (e.g. orbital station) will fail.

oapiGetVesselInterface

Returns the VESSEL class interface for a vessel handle.

Synopsis:

```
VESSEL *oapiGetVesselInterface (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

Pointer to VESSEL class interface for this vessel (see section 11).

oapiGetFocusInterface

Returns the VESSEL class interface for the current focus object.

Synopsis:

```
VESSEL *oapiGetFocusInterface ()
```

Return value:

Pointer to VESSEL class interface for focus object (see section 11).

oapiCreateVessel

Creates a new vessel.

Synopsis:

```
OBJHANDLE  oapiCreateVessel (
     const char *name,
     const char *classname,
     const VESSELSTATUS &status)
```

Parameters:

name	vessel name
classname	vessel class name
status	status parameters

Return value:

handle of the newly created vessel

Notes:

- A configuration file for the specified vessel class must exist in the Config subdirectory.
- This function replaces VESSEL::Create().

oapiDeleteVessel

Deletes an existing vessel.

Synopsis:

```
bool oapiDeleteVessel (
     OBJHANDLE hVessel,
     OBJHANDLE hAlternativeCameraTarget = 0)
```

Parameters:

hVessel	vessel handle
hAlternativeCameraTarget	optional new camera target

Return value:
true if vessel could be deleted.

Notes:
- The current focus object (i.e. the vessel receiving user input) cannot be deleted. Trying to do so will return false.
- If the current camera target is deleted, a new camera target can be provided in hAlternativeCameraTarget. If not specified, the focus object is used as default camera target.

17.2. Object parameters

oapiGetSize
Returns the size (mean radius) of an object.

Synopsis:
```
double oapiGetSize (OBJHANDLE hObj)
```

Parameters:
hObj object handle

Return value:
Object size (mean radius) in meter.

17.3. Object mass functions

oapiGetMass
Returns the mass [kg] of an object. For vessels, this is the total mass, including current fuel and payload mass.

Synopsis:
```
double oapiGetMass (OBJHANDLE hObj)
```

Parameters:
hObj object handle

Return value:
object mass [kg]

oapiGetEmptyMass
Returns empty mass of a vessel, excluding fuel and payload.

Synopsis:
```
double oapiGetEmptyMass (OBJHANDLE hVessel)
```

Parameters:
hVessel vessel handle

Return value:
empty vessel mass [kg]

Notes:
- hVessel must be a vessel handle. Other object types are invalid.
- Do not rely on a constant empty mass. Structural changes (e.g. discarding a rocket stage) will affect the empty mass.
- For multistage configurations, the fuel mass of all currently inactive stages contributes to the empty mass. Only the fuel mass of active stages is excluded.

oapiGetFuelMass

Returns current fuel mass of a vessel.

Synopsis:

```
double oapiGetFuelMass (OBJHANDLE hVessel)
```

Parameters:

hVessel	vessel handle

Return value:

Current fuel mass [kg]

Notes:

- hVessel must be a vessel handle. Other object types are invalid.
- For multistage configurations, this returns the current fuel mass of active stages only.

oapiGetMaxFuelMass

Returns maximum fuel mass of a vessel.

Synopsis:

```
double oapiGetMaxFuelMass (OBJHANDLE hVessel)
```

Parameters:

hVessel	vessel handle

Return value:

Maximum fuel mass [kg]

Notes:

- hVessel must be a vessel handle. Other object types are invalid.
- For multistage configurations, this returns the sum of the max fuel mass of active stages only.

oapiSetEmptyMass

Set the empty mass of a vessel (excluding fuel and payload)

Synopsis:

```
void oapiSetEmptyMass (OBJHANDLE hVessel, double mass)
```

Parameters:

hVessel	vessel handle
mass	empty mass [kg]

Notes:

- Use this function to register structural mass changes, for example as a result of jettisoning a fuel tank, etc.

17.4. Object state vectors

oapiGetGlobalPos

Returns the position of an object in the global reference frame.

Synopsis:

```
void oapiGetGlobalPos (OBJHANDLE hObj, VECTOR3 *pos)
```

Parameters:

hObj	object handle
pos	pointer to vector receiving coordinates

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.
- Units are meters.

oapiGetGlobalVel

Returns the velocity of an object in the global reference frame.

Synopsis:

```
void oapiGetGlobalVel (OBJHANDLE hObj, VECTOR3 *vel)
```

Parameters:

hObj	object handle
vel	pointer to vector receiving velocity data

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.
- Units are meters/second.

oapiGetFocusGlobalPos

Returns the position of the current focus object in the global reference frame.

Synopsis:

```
void oapiGetFocusGlobalPos (VECTOR3 *pos)
```

Parameters:

pos	pointer to vector receiving coordinates

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.0.
- Units are meters.

oapiGetFocusGlobalVel

Returns the velocity of the current focus object in the global reference frame.

Synopsis:

```
void oapiGetFocusGlobalVel (VECTOR3 *vel)
```

Parameters:

vel	pointer to vector receiving velocity data

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.
- Units are meters/second.

oapiGetRelativePos

Returns the distance vector from hRef to hObj in the ecliptic reference frame.

Synopsis:

```
void oapiGetRelativePos (
     OBJHANDLE hObj,
     OBJHANDLE hRef,
     VECTOR3 *pos)
```

Parameters:

hObj	object handle

hRef reference object handle
pos pointer to vector receiving distance data

Notes:
Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

oapiGetRelativeVel
Returns the velocity difference vector of hObj relative to hRef in the ecliptic reference frame.

Synopsis:
```
void oapiGetRelativeVel (
     OBJHANDLE hObj,
     OBJHANDLE hRef,
     VECTOR3 *vel)
```

Parameters:
hObj object handle
hRef reference object handle
vel pointer to vector receiving velocity difference data

Notes:
Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

oapiGetFocusRelativePos
Returns the distance vector from hRef to the current focus object.

Synopsis:
```
void oapiGetFocusRelativePos (OBJHANDLE hRef, VECTOR3 *pos)
```

Parameters:
hRef reference object handle
pos pointer to vector receiving distance data

Notes:
Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

oapiGetFocusRelativeVel
Returns the velocity difference vector of the current focus object relative to hRef.

Synopsis:
```
void oapiGetFocusRelativeVel (OBJHANDLE hRef, VECTOR3 *vel)
```

Parameters:
hRef reference object handle
vel pointer to vector receiving velocity difference data

Notes:
Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

17.5. Surface-relative parameters

oapiGetAltitude
Returns the altitude of a vessel over a planetary surface.

Synopsis:
```
BOOL oapiGetAltitude (OBJHANDLE hVessel, double *alt)
```

Parameters:

hVessel vessel handle
alt pointer to variable receiving altitude value

Return value:
Error flag (FALSE on failure)

Notes:
- Unit is meter [m]
- Returns altitude above *closest* planet.
- Altitude is measured above *mean* planet radius (as defined by SIZE parameter in planet's cfg file)
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusAltitude
Returns the altitude of the current focus vessel over a planetary surface.

Synopsis:
```
BOOL oapiGetFocusAltitude (double *alt)
```

Parameters:
alt pointer to variable receiving altitude value [m]

Return value:
Error flag (FALSE on failure)

oapiGetPitch
Returns a vessel's pitch angle w.r.t. the local horizon.

Synopsis:
```
BOOL oapiGetPitch (OBJHANDLE hVessel, double *pitch)
```

Parameters:
hVessel vessel handle
pitch pointer to variable receiving pitch value

Return value:
Error flag (FALSE on failure)

Notes:
- Unit is radian [rad]
- Returns pitch angle w.r.t. closest planet
- The local horizon is the plane whose normal is defined by the distance vector from the planet centre to the vessel.
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusPitch
Returns the pitch angle of the current focus vessel w.r.t. the local horizon.

Synopsis:
```
BOOL oapiGetFocusPitch (double *pitch)
```

Parameters:
pitch pointer to variable receiving pitch value

Return value:
Error flag (FALSE on failure)

oapiGetBank
Returns a vessel's bank angle w.r.t. the local horizon.

Synopsis:

```
BOOL oapiGetBank (OBJHANDLE hVessel, double *bank)
```

Parameters:

hVessel	vessel handle
bank	pointer to variable receiving bank value

Return value:

Error flag (FALSE on failure)

Notes:

- Unit is radian [rad]
- Returns bank angle w.r.t. closest planet
- The local horizon is the plane whose normal is defined by the distance vector from the planet centre to the vessel.
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusBank

Returns the bank angle of the current focus vessel w.r.t. the local horizon.

Synopsis:

```
BOOL oapiGetFocusBank (double *bank)
```

Parameters:

bank	pointer to variable receiving bank angle [rad]

Return value:

Error flag (FALSE on failure)

oapiGetHeading

Returns a vessel's heading (against geometric north) calculated for the local horizon plane.

Synopsis:

```
BOOL oapiGetHeading (OBJHANDLE hVessel, double *heading)
```

Parameters:

hVessel	vessel handle
heading	pointer to variable receiving heading value [rad]

Return value:

Error flag (FALSE on failure)

Notes:

- Unit is radian [rad] 0=north, $\pi/2$=east, etc.
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusHeading

Returns the heading (against geometric north) of the current focus vessel calculated for the local horizon plane.

Synopsis:

```
BOOL oapiGetFocusHeading (double *heading)
```

Parameters:

heading	pointer to variable receiving heading value [rad]

Return value:

Error flag (FALSE on failure)

oapiGetEquPos

Returns a vessel's spherical equatorial coordinates (longitude, latitude and radius) with respect to the closest planet or moon.

Synopsis:

```
BOOL oapiGetEquPos (
     OBJHANDLE hVessel,
     double *longitude,
     double *latitude,
     double *radius)
```

Parameters:

hVessel	vessel handle
longitude	pointer to variable receiving longitude value [rad]
latitude	pointer to variable receiving latitude value [rad]
radius	pointer to variable receiving radius value [m]

Return value:

Error flag (FALSE on failure)

Notes:

- The handle passed to the function must refer to a vessel; stations are not supported at present.

oapiGetFocusEquPos

Returns the current focus vessel's spherical equatorial coordinates (longitude, latitude and radius) with respect to the closest planet or moon.

Synopsis:

```
BOOL oapiGetFocusEquPos (
     double *longitude,
     double *latitude,
     double *radius)
```

Parameters:

longitude	pointer to variable receiving longitude value [rad]
latitude	pointer to variable receiving latitude value [rad]
radius	pointer to variable receiving radius value [m]

Return value:

Error flag (FALSE on failure)

oapiGetAirspeed

Returns a vessel's airspeed w.r.t. the closest planet or moon.

Synopsis:

```
BOOL oapiGetAirspeed (OBJHANDLE hVessel, double *airspeed)
```

Parameters:

hVessel	vessel handle
airspeed	pointer to variable receiving airspeed value [m/s]

Return value

Error flag (FALSE on failure)

Notes:

- This function works even for planets or moons without atmosphere. It returns an "airspeed-equivalent" value.

oapiGetFocusAirspeed

Returns the current focus vessel's airspeed w.r.t. the closest planet or moon.

Synopsis:
```
BOOL oapiGetFocusAirspeed (double *airspeed)
```

Parameters:
airspeed pointer to variable receiving airspeed value [m/s]

Return value:
Error flag (FALSE on failure)

oapiGetAirspeedVector

Returns a vessel's airspeed vector w.r.t. the closest planet or moon in the local horizon's frame of reference.

Synopsis:
```
BOOL oapiGetAirspeedVector (
     OBJHANDLE hVessel,
     VECTOR3 *speedvec)
```

Parameters:
hVessel vessel handle
speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:
Error flag (FALSE on failure)

Notes:
- This function returns the airspeed vector with respect to the local horizon reference frame. To get the vector with respect to the local vessel coordinates, use oapiGetShipAirspeedVector.

oapiGetFocusAirspeedVector

Returns the current focus vessel's airspeed vector w.r.t. the closest planet or moon in the local horizon's frame of reference.

Synopsis:
```
BOOL oapiGetFocusAirspeedVector (VECTOR3 *speedvec)
```

Parameters:
speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:
Error flag (FALSE on failure)

oapiGetShipAirspeedVector

Returns a vessel's airspeed vector w.r.t. the closest planet or moon in the vessel's local frame of reference.

Synopsis:
```
BOOL oapiGetShipAirspeedVector (
     OBJHANDLE hVessel,
     VECTOR3 *speedvec)
```

Parameters:
hVessel vessel handle
speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:
Error flag (FALSE on failure)

Notes:

- This function returns the airspeed vector with respect to the vessel's frame of reference. The get the vector with respect to the local horizon's frame of reference, use oapiGetAirspeedVector.

oapiGetFocusShipAirspeedVector

Returns the current focus vessel's airspeed vector w.r.t. closest planet or moon in the vessel's local frame of reference.

Synopsis:

```
BOOL oapiGetFocusShipAirspeedVector (VECTOR3 *speedvec)
```

Parameters:

speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:

Error flag (FALSE on failure)

oapiGetAtmPressureDensity

Returns the atmospheric pressure and density caused by a planetary atmosphere at the current vessel position.

Synopsis:

```
void oapiGetAtmPressureDensity (
     OBJHANDLE hVessel,
     double *pressure,
     double *density)
```

Parameters:

hVessel vessel handle
pressure pointer to variable receiving pressure value [Pa]
density pointer to variable receiving density value [kg/m^3]

Notes:

- Pressure and density are calculated using an exponential barometric equation, without accounting for local variations.

oapiGetFocusAtmPressureDensity

Returns the atmospheric pressure and density caused by a planetary atmosphere at the current focus vessel's position.

Synopsis:

```
void oapiGetFocusAtmPressureDensity (
     double *pressure,
     double *density)
```

Parameters:

pressure pointer to variable receiving pressure value [Pa]
density pointer to variable receiving density value [kg/m^3]

17.6. Engine status

oapiGetEngineStatus

Retrieve the status of main, retro and hover thrusters for a vessel.

Synopsis:

```
void oapiGetEngineStatus (
     OBJHANDLE hVessel,
     ENGINESTATUS *es)
```

Parameters:

hVessel	vessel handle
es	pointer to an ENGINESTATUS structure which will receive the engine level parameters

Notes:

The main/retro engine level has a range of [-1,+1]. A positive value indicates engaged main/disengaged retro thrusters, a negative value indicates engaged retro/disengaged main thrusters. Main and retro thrusters cannot be engaged simultaneously. For vessels without retro thrusters the valid range is [0,+1]. The valid range for hover thrusters is [0,+1].

oapiGetFocusEngineStatus

Retrieve the engine status for the focus vessel.

Synopsis:

```
void oapiGetFocusEngineStatus (ENGINESTATUS *es)
```

Parameters:

es	pointer to an ENGINESTATUS structure which will receive the engine level parameters

Notes:

See oapiGetEngineStatus

oapiSetEngineLevel

Engage the specified engines.

Synopsis:

```
void oapiSetEngineLevel (
     OBJHANDLE hVessel,
     ENGINETYPE engine,
     double level)
```

Parameters:

hVessel	vessel handle
engine	identifies the engine to be set
level	engine thrust level [0,1]

Notes:

- Not all vessels support all types of engines.
- Setting main thrusters >0 implies setting retro thrusters to 0 and vice versa.
- Setting main thrusters to –level is equivalent to setting retro thrusters to +level and vice versa.

oapiGetAttitudeMode

Returns a vessel's current attitude thruster mode.

Synopsis:

```
int oapiGetAttitudeMode (OBJHANDLE hVessel)
```

Parameters:

hVessel	vessel handle

Return value:

Current attitude mode (0=disabled or not available, 1=rotational, 2=linear)

Notes:

- The handle must refer to a vessel. This function does not support stations or other object types.

oapiToggleAttitudeMode

Flip a vessel's attitude thruster mode between rotational and linear.

Synopsis:

```
int oapiToggleAttitudeMode (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

The new attitude mode (1=rotational, 2=linear, 0=unchanged disabled)

Notes:

- The handle must refer to a vessel. This function does not support stations or other object types.
- This function flips between linear and rotational, but has no effect if attitude thrusters were disabled.

oapiSetAttitudeMode

Set a vessel's attitude thruster mode.

Synopsis:

```
bool oapiSetAttitudeMode (OBJHANDLE hVessel, int mode)
```

Parameters:

hVessel vessel handle
mode attitude mode (0=disable, 1=rotational, 2=linear)

Return value:

Error flag; *false* indicates failure (requested mode not available)

Notes:

- The handle must refer to a vessel. This function does not support stations or other object types.

oapiGetFocusAttitudeMode

Returns the current focus vessel's attitude thruster mode (rotational or linear)

Synopsis:

```
int oapiGetFocusAttitudeMode ()
```

Return value:

Current attitude mode (0=disabled or not available, 1=rotational, 2=linear)

oapiToggleFocusAttitudeMode

Flip the current focus vessel's attitude thruster mode between rotational and linear.

Synopsis:

```
int oapiToggleFocusAttitudeMode ()
```

Return value:

The new attitude mode (1=rotational, 2=linear, 0=unchanged disabled)

Notes:

- This function flips between linear and rotational, but has no effect if attitude thrusters were disabled.

oapiSetFocusAttitudeMode

Set the current focus vessel's attitude thruster mode.

Synopsis:

```
bool oapiSetFocusAttitudeMode (int mode)
```

Parameters:

mode attitude mode (0=disable, 1=rotational, 2=linear)

Return value:

Error flag; *false* indicates error (requested mode not available)

17.7. Simulation time

oapiGetSimTime

Retrieve time (in seconds) since simulation start.

Synopsis:

```
double oapiGetSimTime ()
```

Return value:

Simulation up time (seconds)

Notes:

Since the simulation up time depends on the simulation start time, this parameter is useful mainly for time differences. To get an absolute time parameter, use oapiGetSimMJD.

oapiGetSimStep

Retrieve length of last time step (from previous to current frame) in seconds.

Synopsis:

```
double oapiGetSimStep ()
```

Return value:

Step length (seconds)

Notes:

This parameter is useful for numerical (finite difference) calculation of time derivatives.

oapiGetSimMJD

Retrieve absolute time measure (Modified Julian Date) for current simulation state.

Synopsis:

```
double oapiGetSimMJD ()
```

Return value:

Current Modified Julian Date (days)

Notes:

Orbiter defines the *Modified Julian Date* (MJD) as JD – 240 0000.5, where JD is the *Julian Date*. JD is the interval of time in mean solar days elapsed since 4713 BC January 1 at Greenwich mean noon.

oapiTime2MJD

Convert a simulation up time value into a Modified Julian Date.

Synopsis:

```
double oapiTime2MJD (double simt)
```

Parameters:

simt simulation time (seconds)

Return value:

Modified Julian Date (MJD) corresponding to simt.

oapiGetTimeAcceleration

Returns simulation time acceleration factor.

Synopsis:

```
double oapiGetTimeAcceleration (void)
```

Return value:

time acceleration factor

Notes:

This function will not return 0 when the simulation is paused. Instead it will return the acceleration factor at which the simulation will resume when unpaused.

oapiSetTimeAcceleration

Set the simulation time acceleration factor

Synopsis:

```
void oapiSetTimeAcceleration (double warp)
```

Parameters:

warp new time acceleration factor

Notes:

Warp factors will be clamped to the valid range [1,1000]. If the new warp factor is different from the previous one, all DLLs (including the one that called oapiSetTimeAcceleration) will be sent a `opcTimeAccChanged` message.

17.8. Keyboard input

oapiAcceptDelayedKey

This function is obsolete and should no longer be used. See ovcConsumeBufferedKey for handling buffered key events. *May be removed in a future version.*

17.9. Mesh management

oapiLoadMesh

Loads a mesh from file and returns a handle to it.

Synopsis:

```
MESHHANDLE oapiLoadMesh (const char *fname)
```

Parameters:

fname mesh file name

Return value:

Handle to the loaded mesh. (NULL indicates load error)

Notes:

- The file name should not contain a path or file extension. Orbiter appends extension .msh and searches in the default mesh directory.
- Meshes should be deallocated with oapiDeleteMesh() when no longer needed.

See also:
oapiDeleteMesh(), VESSEL::AddMesh()

oapiLoadMeshGlobal

Retrieves a mesh handle from the global mesh manager. When called for the first time for any given file name, the mesh is loaded from file and stored as a system resource. Every further request of the same mesh directly returns a handle to the stored mesh without further file I/O.

Synopsis:
```
const MESHHANDLE oapiLoadMeshGlobal (const char *fname)
```

Parameters:
fname mesh file name

Return value:
mesh handle

Notes:
- Once a mesh is globally loaded it remains in memory until the user closes the simulation window.
- This function can be used to pre-load meshes to avoid load delays during the simulation. For example, parent objects may pre-load meshes for any child objects they may create later.
- Do *NOT* delete any meshes obtained by this function with oapiDeleteMesh! Orbiter takes care of deleting globally managed meshes.

oapiDeleteMesh

Removes a mesh from memory.

Synopsis:
```
void oapiDeleteMesh (MESHHANDLE hMesh)
```

Parameters:
hMesh mesh handle

17.10. HUD, Panel and MFD management

oapiSetHUDMode

Set HUD (head up display) mode.

Synopsis:
```
bool oapiSetHUDMode (int mode)
```

Parameters:
mode new HUD mode

Return value:
true if mode has changed, false otherwise.

Notes:
- Mode HUD_NONE will turn off the HUD display.
- See constants HUD_xxx (section 10) for currently supported HUD modes.

oapiGetHUDMode

Query current HUD (head up display) mode.

Synopsis:
int oapiGetHUDMode ()

Return value:
Current HUD mode

oapiOpenMFD
Set an MFD (multifunctional display) to a specific mode.

Synopsis:
```
void oapiOpenMFD (int mode, int id)
```

Parameters:
mode	MFD mode (see Section 10)
id	MFD identifier (see Section 10)

Notes:
- mode `MFD_NONE` will turn off the MFD.
- For the on-screen instruments, only `MFD_LEFT` and `MFD_RIGHT` are supported. Custom panels may support (up to 3) additional MFDs.

oapiGetMFDMode
Get the current mode of the specified MFD.

Synopsis:
```
int oapiGetMFDMode (int id)
```

Parameters:
id	MFD identifier (see Section 10)

Return value:
MFD mode (see Section 10)

oapiSendMFDKey
Sends a keystroke to an MFD.

Synopsis:
```
int oapiSendMFDKey (int id, DWORD key)
```

Parameters:
id	MFD identifier (see Section 10)
key	key code (see OAPI_KEY_xxx constants in orbitersdk.h)

Return value:
nonzero if the MFD understood and processed the key.

Notes:
- This function can be used to interact with the MFD as if the user had pressed Shift-*key*, for example to select a different MFD mode, to select a target body, etc.

oapiProcessMFDButton
Requests a default action as a result of a MFD button event.

Synopsis:
```
virtual bool ProcessMFDButton (
     int mfd,
     int bt,
     int event) const
```

Parameters:
mfd	MFD identifier (see Section 10)

bt button number (≥ 0)
event mouse event (a combination of PANEL_MOUSE_xxx flags)

Return value:
Returns true if the button was processed, false if no action was assigned to the button.

Notes:
- Orbiter assigns default button actions for the various MFD modes. For example, in *Orbit* mode the action assigned to button 0 is *Select reference*. Calling oapiProcessMFDButton (for example as a reaction to a mouse button event) will execute this action.

oapiMFDButtonLabel
Retrieves a default label for an MFD button.

Synopsis:
```
const char *oapiMFDButtonLabel (int mfd, int bt)
```

Parameters:
mfd MFD identifier (see Section 10)
bt button number (≥ 0)

Return value:
pointer to static string containing the label, or NULL if the button is not assigned.

Notes:
- Labels contain 1 to 3 characters.
- This function can be used to paint the labels on the MFD buttons of a custom panel.
- The labels correspond to the default button actions executed by VESSEL::ProcessMFDButton.

oapiRegisterMFD
Registers an MFD position for a custom panel.

Synopsis:
```
void oapiRegisterMFD (int id, const MFDSPEC &spec)
```

Parameters:
id MFD identifier (see Section 10)
spec MFD parameters (see below)

Notes:
- Should be called in the body of `ovcLoadPanel` for panels which define MFDs.
- Defining more than 2 or 3 MFDs per panel can degrade performance.
- MFDSPEC is a struct with the following fields:

```
typedef struct {
  RECT pos;         position of MFD in panel (pixel)
  int nbt_left;     number of buttons on left side of MFD display
  int nbt_right;    number of buttons on right side of MFD display
  int bt_yofs;      y-offset of top button from top display edge (pixel)
  int bt_ydist;     y-distance between buttons (pixel)
} MFDSPEC;
```

oapiRegisterPanelBackground
Register the background bitmap for a custom panel.

Synopsis:

```
void oapiRegisterPanelBackground (
     HBITMAP hBmp,
     DWORD flag = PANEL_ATTACH_BOTTOM|PANEL_MOVEOUT_BOTTOM,
     DWORD ck = (DWORD)-1)
```

Parameters:

hBmp	bitmap handle
flag	property bit flags (see notes)
ck	transparency colour key

Notes:

- This function will normally be called in the body of ovcLoadPanel.
- Typically the bitmap will be stored as a resource in the DLL and obtained by a call to the Windows function LoadBitmap(...).
- flag defines panel properties and can be a combination of the following bitmasks:
  PANEL_ATTACH_{LEFT/RIGHT/TOP/BOTTOM}
  PANEL_MOVOUT_{LEFT/RIGHT/TOP/BOTTOM}
  where PANEL_ATTACH_BOTTOM means that the bottom edge of the panel cannot be scrolled above the bottom edge of the screen (other directions work equivalently) and PANEL_MOVEOUT_BOTTOM means that the panel can be scrolled downwards out of the screen (other directions work equivalently)
- The colour key, if defined, specifies a colour which will appear transparent when displaying the panel. The key is in (hex) 0xRRGGBB format. If no key is specified, the panel will be opaque. It is best to use black (0x000000) or white (0xffffff) as colour keys, since other values may cause problems in 16bit screen modes. Of course, care must be taken that the keyed colour does not appear anywhere in the opaque part of the panel.

oapiRegisterPanelArea

Defines a rectangular area within a panel to receive mouse or redraw notifications.

Synopsis:

```
void oapiRegisterPanelArea (
     int aid,
     const RECT &pos,
     int draw_event = PANEL_REDRAW_NEVER,
     int mouse_event = PANEL_MOUSE_IGNORE,
     int bkmode = PANEL_MAP_NONE)
```

Parameters:

aid	area identifier
pos	bounding box of the marked area
draw_event	defines redraw events
mouse_event	defines mouse events
bkmode	redraw background mode

Notes:

- Each panel area must be defined with an identifier *aid* which is unique within the panel.
- draw_event can have the following values:
  PANEL_REDRAW_NEVER: do not generate redraw events.
  PANEL_REDRAW_ALWAYS: generate a redraw event at every time step.
  PANEL_REDRAW_MOUSE: mouse events trigger redraw events.
- For possible values of mouse_event see orbitersdk.h.
  PANEL_MOUSE_IGNORE prevents mouse events from being triggered.

- bkmode defines the bitmap handed to the redraw callback:
  PANEL_MAP_NONE: provides an undefined bitmap. Should be used if the whole area is repainted.
  PANEL_MAP_CURRENT: provides a copy of the current area.
  PANEL_MAP_BACKGROUND: provides a copy of the panel background (as defined by oapiRegisterPanelBackground).

oapiSetPanelNeighbours

Defines the neighbour panels of the current panels. These are the panels the user can switch to via Ctrl-Arrow keys.

Synopsis:

```
void oapiSetPanelNeighbours (
     int left,
     int right,
     int top,
     int bottom)
```

Parameters:

left	panel id of left neighbour (or –1 if none)
right	panel id of right neighbour (or –1 if none)
top	panel id of top neighbour (or –1 if none)
bottom	panel id of bottom neighbour (or –1 if none)

Notes:

- This function should be called during panel registration (in ovcLoadPanel) to define the neighbours of the registered panel.
- Every panel (except panel 0) must be listed as a neighbour by at least one other panel, otherwise it is inaccessible.

oapiTriggerPanelRedrawArea

Triggers a redraw notification for a panel area.

Synopsis:

```
void oapiTriggerPanelRedrawArea (int panel_id, int area_id)
```

Parameters:

panel_id	panel identifier (≥0)
area_id	area identifier (≥0)

Notes:

- The redraw notification is ignored if the requested panel is not currently displayed.

oapiGetDC

Obtain a Windows device context handle (HDC) for a surface.

Synopsis:

```
HDC oapiGetDC (SURFHANDLE surf)
```

Parameters:

surf	surface handle

Return value:

device context handle for the surface

Notes:

- The device context can be used to perform standard Windows drawing operations (such as LineTo(), Rectangle(), TextOut(), etc.) on the surface.

- When the context is no longer needed it must be released with a call to oapiReleaseDC.

oapiReleaseDC

Release a previously acquired device context for a surface.

Synopsis:

```
void oapiReleaseDC (SURFHANDLE surf, HDC hDC)
```

Parameters:

surf	surface handle
hDC	device context to be released

Notes:

- Use this function to release a device context previously acquired with oapiGetDC.
- Standard Windows device context rules apply. For example, any custom device objects loaded via SelectObject must be unloaded before calling oapiReleaseDC.

oapiGetColour

Returns a colour value adapted to the current screen colour depth for given red, green and blue components.

Synopsis:

DWORD oapiGetColour (DWORD red, DWORD green, DWORD blue)

Parameters:

red	red component (0-255)
green	green component (0-255)
blue	blue component (0-255)

Return value

colour value

Notes:

- Colour values are required for some surface functions like oapiClearSurface() or oapiSetSurfaceColourKey(). The colour key for a given RGB triplet depends on the screen colour depth. This function returns the colour value for the closest colour match which can be displayed in the current screen mode.
- In 24 and 32 bit modes the requested colour can always be matched. The colour value in that case is (red << 16) + (green << 8) + blue.
- For 16 bit displays the colour value is calculated as ((red*31)/255) << 11 + ((green*63)/255 << 5 + (blue*31)/255 assuming a “565” colour mode (5 bits for red, 6, for green, 5 for blue). This means that a requested colour may not be perfectly matched.
- These colour values should not be used for Windows (GDI) drawing functions where a COLORREF value is expected.

oapiCreateSurface (1)

Create a surface of the specified dimensions.

Synopsis:

```
SURFHANDLE oapiCreateSurface (int width, int height)
```

Parameters:

width	width of surface bitmap (pixels)
height	height of surface bitmap (pixels)

Return value

Handle to the new surface.

Notes:

- The bitmap contents are undefined after creation, so the surface must be repainted fully before mapping it to the screen.

See also:

oapiDestroySurface()

oapiCreateSurface (2)

Create a surface from a bitmap. Bitmap surfaces are typically used for blitting operations during instrument panel redraws.

Synopsis:

```
SURFHANDLE oapiCreateSurface (
     HBITMAP hBmp,
     bool release_bmp = true)
```

Parameters:

hBmp	bitmap handle
release_bmp	flag for bitmap release

Return value:

Handle to the new surface.

Notes:

- The easiest way to access bitmaps is by storing them as resources in the module, and loading them via a call to LoadBitmap().
- Do not use this function with a bitmap generated by CreateBitmap(). To create a surface of specified dimensions, use oapiCreateSurface (width, height) instead.
- If release_bmp == true, then oapiCreateSurface will destroy the bitmap after creating a surface from it (i.e. the hBmp handle will be invalid after the function returns), otherwise the module is responsible for destroying the bitmap by a call to DestroyObject() when it is no longer needed.
- Surfaces should be destroyed by calling oapiDestroySurface() when they are no longer needed.

oapiDestroySurface

Destroy a surface previously created with oapiCreateSurface.

Synopsis:

```
void oapiDestroySurface (SURFHANDLE surf)
```

Parameters:

surf	surface handle

oapiSetSurfaceColourKey

Define a colour key for a surface to allow transparent blitting.

Synopsis:

```
void oapiSetSurfaceColourKey (SURFHANDLE surf, DWORD ck)
```

Parameters:

surf	surface handle
ck	colour key (0xRRGGBB)

Notes:

- Defining a colour key and subsequently calling oapiBlt with the SURF_PREDEF_CK flag is slightly more efficient than passing the colour key explicitly to oapiBlt each time, if the same colour key is used repeatedly.

See also:
oapiClearSurfaceColourKey(), oapiBlt()

oapiClearSurfaceColourKey
Clear a previously defined colour key.

Synopsis:
```
void oapiClearSurfaceColourKey (SURFHANDLE surf)
```

Parameters:

surf	surface handle

See also:
oapiSetSurfaceColourKey(), oapiBlt()

oapiBlt
Copy a surface into another surface.

Synopsis:
```
void oapiBlt (
     SURFHANDLE tgt, SURFHANDLE src,
     int tgtx, int tgty,
     int srcx, int srcy,
     int w, int h,
     DWORD ck = SURF_NO_CK)
```

Parameters:

tgt	target surface
src	source surface
tgtx, tgty	coordinates of upper left corner of copied area in target bitmap.
srcx, srcy	coordinates of upper left corner of copied area in source bitmap.
w, h	width, height of copied rectangle (pixel)
ck	colour key (see notes)

Notes:
- Typically, this function is used to update panel instruments during processing of ovcPanelRedrawEvent.
- This function must not be used while a device context is acquired for the target surface (i.e. between oapiGetDC and oapiReleaseDC calls).
- If a blitting operation is necessary between oapiGetDC and oapiReleaseDC, you may use the standard Windows BitBlt function. However this does not use hardware acceleration and should therefore be avoided.
- Transparent blitting can be performed by specifying a colour key in ck. The transparent colour can either be passed explicitly in ck, or ck can be set to SURF_PREDEF_CK to use the key previously defined with oapiSetSurfaceColourKey().

See also:
oapiSetSurfaceColourKey()

17.11. Custom MFD modes

oapiRegisterMFDMode
Register a custom MFD mode.

Synopsis:
```
int oapiRegisterMFDMode (MFDMODESPEC &spec)
```

Parameters:

spec MFD specs (see notes below)

Return value:

MFD mode identifier

Notes:

- This function registers a custom MFD mode with Orbiter. There are two types of custom MFDs: generic and vessel class-specific. Generic MFD modes are available to all vessel types, while specific modes are only available for a single vessel class. Generic modes should be registered in the `opcDLLInit` callback function of a plugin module. Vessel class specific modes are not implemented yet.
- MFDMODESPEC is a struct defining the parameters of the new mode:

```
typedef struct {
  char *name;       points to the name of the new mode
  int (*msgproc)(UINT,UINT,WPARAM,LPARAM);
                    address of MFD message parser
} MFDMODESPEC;
```

- See orbitersdk\samples\CustomMFD for a sample MFD mode implementation.

oapiUnregisterMFDMode

Unregister a previously registered custom MFD mode.

Synopsis:

```
bool oapiUnregisterMFDMode (int mode)
```

Parameters:

mode mode identifier, as returned by RegisterMFDMode

Return value:

true on success (mode could be unregistered).

17.12. File management

oapiWriteLine

Writes a line to a file.

Synopsis:

```
void oapiWriteLine (FILEHANDLE file, char *line)
```

Parameters:

file file handle
line line to be written (zero-terminated)

oapiWriteScenario_string

Writes a string-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_string (
     FILEHANDLE scn,
     char *item,
     char *string)
```

Parameters:

scn file handle
item item id
string string to be written (zero-terminated)

oapiWriteScenario_int

Writes an integer-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_int (
     FILEHANDLE scn,
     char *item,
     int i)
```

Parameters:

scn	file handle
item	item id
i	integer value to be written

oapiWriteScenario_float

Writes a floating point-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_float (
     FILEHANDLE scn,
     char *item,
     double d)
```

Parameters:

scn	file handle
item	item id
d	floating point value to be written

oapiWriteScenario_vec

Writes a vector-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_vec (
     FILEHANDLE scn,
     char *item,
     const VECTOR3 &vec)
```

Parameters:

scn	file handle
item	item id
vec	vector to be written

oapiReadScenario_nextline

Reads an item from a scenario file.

Synopsis:

```
bool oapiReadScenario_nextline (
     FILEHANDLE scn,
     char *&line)
```

Parameters:

scn	file handle
line	pointer to the scanned line

Notes:

- The function returns true as long as an item for the current block could be read. It returns false at EOF, or when an “END” token is read.
- Leading and trailing whitespace, and trailing comments (from “;” to EOL) are automatically removed.

- "line" points to an internal static character buffer.

17.13. User input

oapiOpenInputBox

Opens a modal input box requesting a string from the user.

Synopsis:

```
void oapiOpenInputBox (
     char *title,
     bool (*Clbk)(void*,char*,void*),
     char *buf = 0,
     int vislen = 20,
     void *usrdata = 0)
```

Parameters:

title	input box title
Clbk	callback function receiving the result of the user input (see notes)
buf	initial state of the input string
vislen	number of characters visible in input box
usrdata	user-defined data passed to the callback function

Notes:

- Format for callback function:
  `bool InputCallback (void *id, char *str, void *usrdata)`
  where *id* identifies the input box, *str* contains the user-supplied string, and *usrdata* contains the data specified in the call to oapiOpenInputBox.
  The callback function should return true if it accepts the string, false otherwise (the box will not be closed if the callback function returns false).
- The box can be closed by the user by pressing Enter ("OK") or Esc ("Cancel"). The callback function is only called in the first case.
- The input box is modal, i.e. all keyboard input is redirected into the dialog box. Normal key functions resume after the box is closed.

17.14. Debugging

oapiDebugString

Returns a pointer to a string which will be displayed in the lower left corner of the viewport.

Synopsis:

```
char *oapiDebugString ()
```

Return value:

Pointer to debugging string.

Notes:

- This function should only be used for debugging purposes. Do not use it in published modules!
- The returned pointer refers to a global char[256] in the Orbiter core. It is the responsibility of the module to ensure that no overflow occurs.
- If the string is written to more than once per time step (either within a single module or by multiple modules) the last state before rendering will be displayed.
- A typical use would be:

```
sprintf (oapiDebugString(), "my value is %f", myvalue);
```

18. Standard ORBITER modules

18.1. Vsop87

Vsop87.dll is a full implementation of the VSOP87 planetary solutions for Mercury to Neptune.[1] Orbiter uses the VSOP87 "B" series which computes the heliocentric positions for the ecliptic and equinox of J2000. Positions and velocities are calculated by a perturbation method which uses a series of trigonometric perturbation terms. The number of included terms defines the precision of the result. Therefore the computation time will depend on the selected precision. Vsop87.dll supports precision settings between 1e-3 and 1e-8.

Vsop87.dll supports the following planets: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune.

According to the VSOP documentation, at full precision (1e-8), the relative error is within 1'' for

- Mercury, Venus, Earth and Mars over 4000 years before and after J2000
- Jupiter and Saturn over 2000 years before and after J2000.
- Uranus and Neptune over 6000 years before and after J2000.

If you want to replace Vsop87 with your own code:

- Check section 15 for the callback interface.
- The code for different planets doesn't need to be implemented in a single DLL. You can replace the calculations for a single planet by writing a module for it, and referencing this module from the planet's cfg file, while keeping the standard Vsop87 module for the other planets.

18.2. Luna

Luna.dll calculates lunar positions and velocities using a perturbation method. The implementation is derived from Elwood Downey's xephem,[2] with time derivative terms added by myself. Coordinates are for equinox and ecliptic of date.

19. Index

W

[1] P. Bretagnon (pierre@bdl.fr) and G. Francou (francou@bdl.fr), Bureau des Longitudes, CNRS URA 707, Planetary Solution VSOP87

[2] Elwood Downey, www.clearskyinstitute.com/xephem/xephem.html